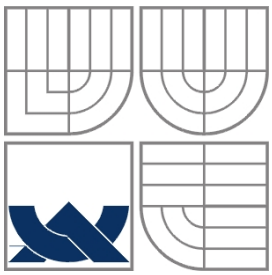
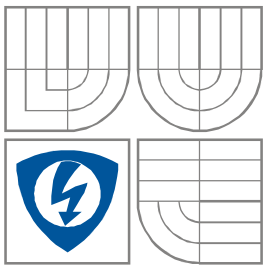


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF CONTROL AND INSTRUMENTATION

ŘÍZENÍ MODELU JEŘÁBU KOMUNIKUJÍCÍHO PROTOKOLEM MODBUS TCP

GANTRY CRANE CONTROL VIA MODBUS TCP

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAKUB ARM

VEDOUCÍ PRÁCE
SUPERVISOR

ING. VÁCLAV KACZMARCZYK

BRNO 2011

ORIGINÁLNÍ ZADÁNÍ DIPLOMOVÉ / BAKALÁŘSKÉ PRÁCE

Abstrakt

Modbus TCP je protokol pro komunikaci po ethernetovém rozhraní. Cílem této práce je realizace řízení daného modelu jeřábu z osobního počítače pomocí daného zařízení komunikujícího protokolem Modbus TCP přes ethernetové rozhraní. Řízení je realizováno z uživatelské aplikace naprogramované v C# a z webového rozhraní běžícího na vytvořeném HTTP serveru. Oboje rozhraní využívá vytvořenou knihovnu, která je naprogramována v jazyce C a komunikují přes Modbus TCP se zařízením Modicon Momentum, které interpretuje příkazy Modbus protokolu na standardní průmyslové vstupy a výstupy pro procesní instrumentaci, která ovládá model jeřábu.

Klíčová slova

Modbus TCP, HTTP server, DLL knihovna, řízení přes TCP, bakalářská práce, VUT Brno.

Abstract

Modbus TCP is a protocol for communication via ethernet bus. Goal of this work is realization of control of gantry crane model from personal computer. It is accomplished by Modbus converter, which communicates via ethernet bus. There are two ways of controlling. By graphic user application written in C# or by web pages running on created HTTP server. All services use created library, which is written in C language and communicates via Modbus TCP with Modicon Momentum unit. This unit interprets orders in form of Modbus protocol to controlling of standard industry inputs and outputs for process instrumentation. This process instrumentation is binded to control the gantry crane model.

Keywords

Modbus TCP, HTTP server, DLL library, control via TCP, bachelor’s thesis, VUT Brno.

Bibliografická citace:

ARM, J. *Řízení modelu jeřábu komunikujícího protokolem Modus TCP*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2011. 55s. Vedoucí bakalářské práce byl Ing. Václav Kaczmarczyk.

Prohlášení

„Prohlašuji, že svou bakalářskou práci na téma Řízení modelu jeřábu komunikujícího protokolem modus TCP jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.“
Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne: **15. května 2011** ……
podpis autora

Poděkování

Děkuji vedoucímu bakalářské práce Ing. Václavu Kaczmarczykovi za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé bakalářské práce.

V Brně dne: **15. května 2011** ……
podpis autora

Obsah

1 ÚVOD

Řízení hraje v dnešní době automatizace procesů důležitou roli. V současné době je k dispozici mnoho typů a provedení řídících systémů, ale vždy záleží na dané konkrétní aplikaci, protože každý se hodí do jiné aplikace v závislosti na požadavcích, které na řízení klademe, konkrétních podmínkách jako jsou např. různá omezení a v neposlední řadě na ceně.

Způsobů řízení existuje mnoho, v základu se dělí na distribuované a tzv. malou automatizaci. Toto dělení se zakládá na rozmístění rozhodovacích prvků v řídícím systému, způsobem řízení a jeho rozsahem. Malou automatizaci tvoří řízení průmyslovými mikropočítači či programovatelnými logickými automaty, které jsou případně spojeny pomocí komunikační sběrnice do řídícího systému. Distribuované systémy jsou mnohem komplexnější a založeny obecně na koncepci hlavních řídících prvků (počítačů), které pomocí sběrnice komunikují s výkonnou složkou systému, což jsou řadiče konkrétní procesní instrumentace.

Jelikož jsou řízené procesy, a tedy požadavky na řídící systémy stále složitější, je třeba klást větší důraz na přehlednost zobrazení stavu procesu a řízení pro obsluhu, popř. zpřístupnit toto zobrazení více obsluhujícím osobám, aby se minimalizoval lidský faktor při řízení daného procesu. Tato potřeba je řešena SCADA systémy či propojení řídících prvků do průmyslové sítě, popř. její napojení na server. V této práci ale využijeme jednoduchého HTTP serveru, pomocí kterého budeme zobrazovat stav a obsluhovat model.

Modbus TCP je protokol pro komunikaci po ethernetovém rozhraní. Byl vyvinut firmou Modicon, původně pro komunikaci po sériové lince. Jelikož byl vyvinut jako jeden z prvních, byl volně k dispozici a je robustní, byl ostatními výrobci často používán. Tímto se stal rozšířeným komunikačním protokolem v průmyslové praxi pro komunikaci nejrůznějších zařízení spojených do sítě.

1.1 Cíl práce

Cílem práce je realizování řízení daného modelu jeřábu pomocí běžného osobního počítače. Počítač bude spojen s ovladačem procesní instrumentace Modicon Momentum 170 ENT 110 00 ethernetovým rozhraním a bude komunikovat protokolem Modbus TCP. Osobní počítač s operačním systémem Windows není standardně vybaven knihovnami pro tento způsob komunikace. Proto musíme nejdříve vytvořit tyto knihovny v programovacím jazyce C. Dále vytvoříme grafickou uživatelskou aplikaci v C#, která bude tyto knihovny využívat. To vyžaduje vytvořit soubor rozhraní mezi prostředím C# a knihovnou. Nakonec ještě vytvoříme HTTP server, který bude zobrazovat stav modelu a ovládat jej. Tento server poběží paralelně s uživatelskou aplikací a bude s ní sdílet data o stavu modelu. Výstupem tedy bude knihovna pro

komunikaci protokolem Modbus TCP, rozhraní pro použití knihovny v prostředí C#, uživatelská grafická aplikace pro ovládání modelu, HTTP server podporující práci s modelem, WWW stránky pro zobrazování a ovládání modelu pomocí knihovny.

1.2 Význam

Práce má za úkol demonstrovat možnosti využití osobního počítače v řízení strojů a zařízení, a to hlavně pomocí ethernetu protokolem Modus TCP. Díky tomuto protokolu se komunikace po ethernetu mezi řídícím počítačem a jednotlivými zařízeními stává pseudodeterministická. V síti existuje jeden řídící prvek, který může náhodně přistupovat na sběrnici, a díky protokolu musí co nejdříve dostat na poslanou žádost danému zařízení odpověď, na kterou čeká. Tímto se komunikace stává řízená. Rychlost komunikace je pak hlavně ovlivněna časem, za který je dané zařízení schopno odeslat odpověď.

Na sběrnici je samozřejmě možné připojit více zařízení, jelikož se jedná o komunikaci v síti. Aby byla zachována řízenost komunikace na sběrnici, je nutné, aby v jeden okamžik využívalo sběrnici pouze jedno řídící zařízení, které posílá žádosti. Za přítomnosti více řízených zařízení se potom při komunikaci se všemi zařízením postupně rychlost komunikace snižuje úměrně k počtu připojených řízených zařízení.

2 MODBUS TCP

Zařízení, která řídí jakýkoliv proces, mohou být spojena sběrnicí a komunikovat mezi sebou za účelem jakékoliv výměny informací. Firma Modicon jako jedna z prvních vyvinula komunikační protokol, který používala všechna zařízení vyráběné touto firmou. Tento protokol umožňuje komunikaci zařízení po různých druzích sběrnic (EIA/TIA - 232 - E, EIA - 422, EIA/TIA - 485 - A, ethernet, FM) a tvoří aplikační vrstvu ISO/OSI modelu. Jelikož je tento protokol z licenčního hlediska otevřený, používaly ho ve svých zařízení i jiné firmy, a tím se stal rozšířeným používaným komunikačním průmyslovým protokolem [4]. Tento protokol definuje strukturu zprávy (tzv. rámec), popisuje proces, kdy zařízení vysílá požadavek k jinému, a jakou odpověď toto zařízení pošle zpět. Z hlediska bezpečnosti disponuje definice protokolu nástrojem k zachycení různých chyb komunikace na předcházejících vrstvách RM ISO/OSI pomocí redundantní kontroly. Také definuje strukturu datového modelu na straně zařízení, který reprezentuje konkrétní vstupy, výstupy a registry zařízení.

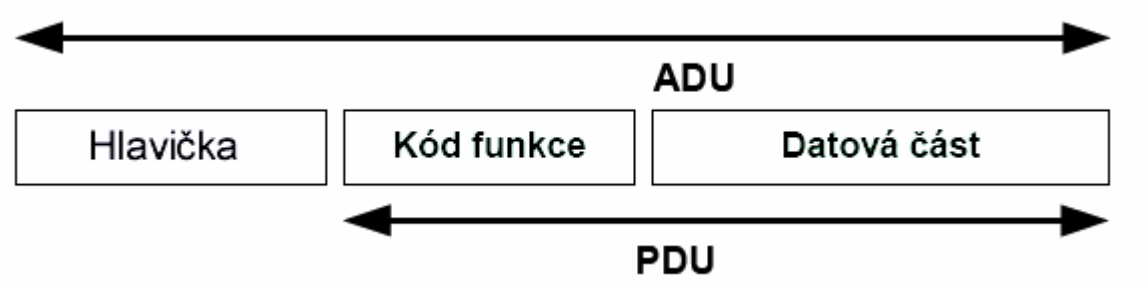
V této práci se budeme zabývat protokolem Modbus TCP, což je protokol Modbus upravený pro komunikaci přes rozhraní ethernet. Tento protokol umožňuje komunikaci mezi zařízeními typu Client/Server. Je nadřazen protokolu TCP, který se stará o bezpečný přenos na úrovni transportní vrstvy referenčního modelu ISO/OSI. Ten zapouzdřuje protokol IP, který se stará o směrování v síti na úrovni síťové vrstvy. Na úrovni linkové vrstvy je k dispozici protokol CSMA/CD, který se stará o řízení přístupu k médiu a zabraňuje kolizím, jelikož může v každém okamžiku vysílat právě jakékoliv zařízení.

2.1 Zpráva

Komunikace mezi zařízeními spočívá ve výměně dat, resp. paketů. Paket obsahuje informace o zdroji, o cíli, vlastní data a jiné další informace. Na aplikační vrstvě mezi sebou zařízení komunikují pomocí zpráv, jejichž struktura je dána protokolem.

2.1.1 Popis zprávy

Zařízení mezi sebou komunikují prostřednictvím výměny zpráv (frame) nesoucí informace, jejichž struktura je složena z části definované protokolem (PDU - Protocol Data Unit) a části, která závisí na použitém nosném rozhraní. Celek je potom platná zpráva na aplikační úrovni (ADU - Application Data Unit). Uspořádání zprávy je zachyceno na obrázku a popsáno dále.



Obr. 2-1 Blokový model Modbus zprávy [2]

Část zprávy závislá na použitém rozhraní (ethernet) se nazývá hlavička (Header) a slouží k definování zprávy v rámci komunikace TCP/IP na aplikační úrovni. Skládá se z čísla zprávy, verze protokolu, identifikace cílového zařízení a délky rámce.

Část zprávy definovaná protokolem (PDU) se skládá z funkčního kódu o délce 1 byte a datového bloku o nespecifikované délce. Funkční kód definuje typ akce, kterou má server provést, či typ chyby, která nastala. Povolené hodnoty jsou 1..255, přičemž 128..255 je vyhrazeno pro chybové odpovědi.

Datový blok nese různé dodatečné informace k vykonání akce, např. adresy, počet a délku datového bloku. V případě, že akce je plně popsaná funkčním kódem, může mít datový blok nulovou délku.

Délka zprávy je z historických důvodů (první implementace Modbus protokolu byla přes sběrnici RS - 485) omezena na 260 bytů, kdy 7 bytů zabírají aplikační informace, které závisí na použitém rozhraní.

Protokol definuje 3 možné typy zpráv : žádost, odpověď, chybová odpověď.

Zprávy žádosti (mb_req_pdu) se skládají z 1 bytového funkčního kódu a několika bytového bloku informací pro zpracování žádosti

Zprávy odpovědi (mb_rsp_pdu) se skládají z 1 bytového funkčního kódu a několika bytového datového bloku odpovědi

Zprávy chybové odpovědi (mb_excep_rsp_pdu) se skládají z 1 bytového funkčního kódu sečteného s 0x80 a 1 bytového definovaného chybového kódu.

Protokolem definovaný způsob odesílání bytových posloupností reprezentující data je "big - Endian", to znamená, že odeslání čísla delšího jak 8 bitů se provede tak, že se nejdříve odešlou byty s větší váhou. Na úrovni bytu se bity posílají metodou MSB First, tedy bit s největší váhou jako první.

Následující tabulka popisuje rámec (požadavek), který je zaslán hostiteli. Jedná se tedy o aplikační zprávu (ADU) obsahující náležitosti Modbus protokolu a část definovanou pro ethernetové rozhraní.

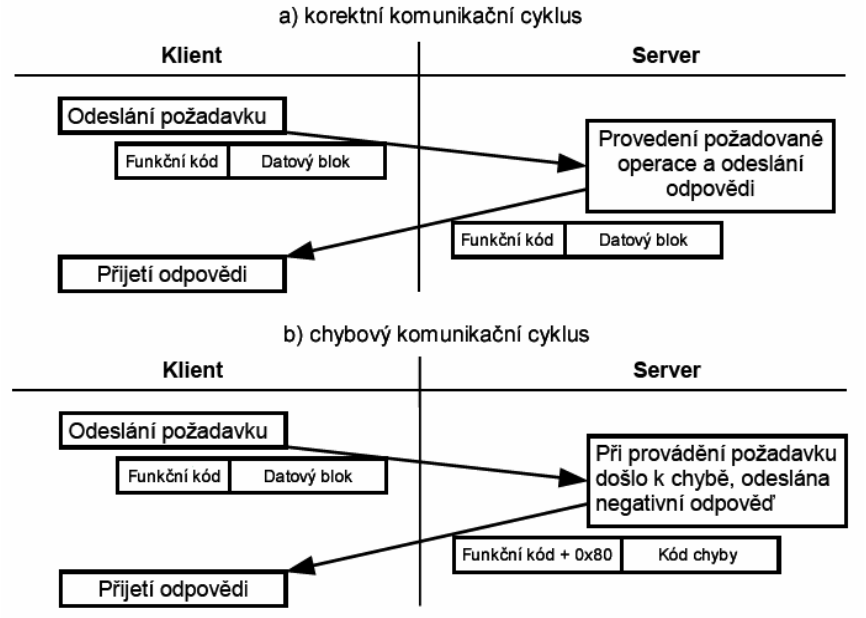
Název bloku	**Délka**	**Popis**
Identifikátor	2 byty	Libovolné číslo definující konkrétní transakci
Verze protokolu	2 byty	Vždy 0x00, 0x00

Délka rámce	2 byty	Počet bytů za tímto blokem
Adresa zařízení	1 byte	Adresa cílového zařízení
Kód funkce	1 byte	Typ zprávy či výjimky
Počáteční adresa	2 byty	Počáteční pracovní adresa hostitele
Počet dat	2 byty	Určuje pracovní délku dat
Délka dat	1 byte	Délka dat za tímto blokem
Data	N bytů	Data pro zpracování zprávy

Tab. 2-1 Struktura Modbus zprávy

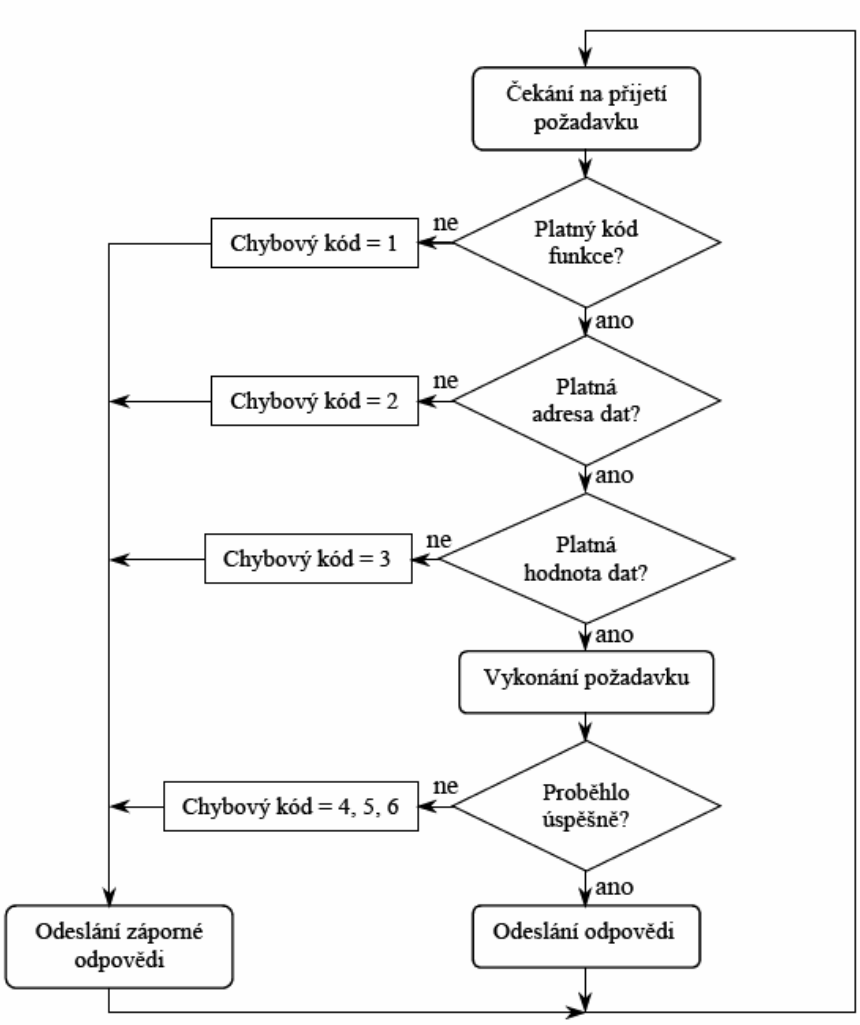
2.1.2 Transakce zprávy

Vlastní komunikace (výměna zpráv) probíhá podle modelu client/server. To znamená, že server čeká na požadavek od klienta, ten pošle tedy zprávu typu žádost se všemi náležitostmi. Po přijetí server testuje, zdali je příchozí zpráva korektní, to znamená, zda má platný kód funkce, platnou adresaci a hodnotu dat. Potom požadavek vykoná. V případě korektního průběhu pošle zpět klientovi zprávu typu odpověď se stejným funkčním kódem, jakou měla žádost. V opačném případě pošle klientovi zprávu typu chybová odpověď s kódem chyby. Tímto cyklus komunikace skončí.



Obr. 2-2 Schéma komunikačních cyklů [1]

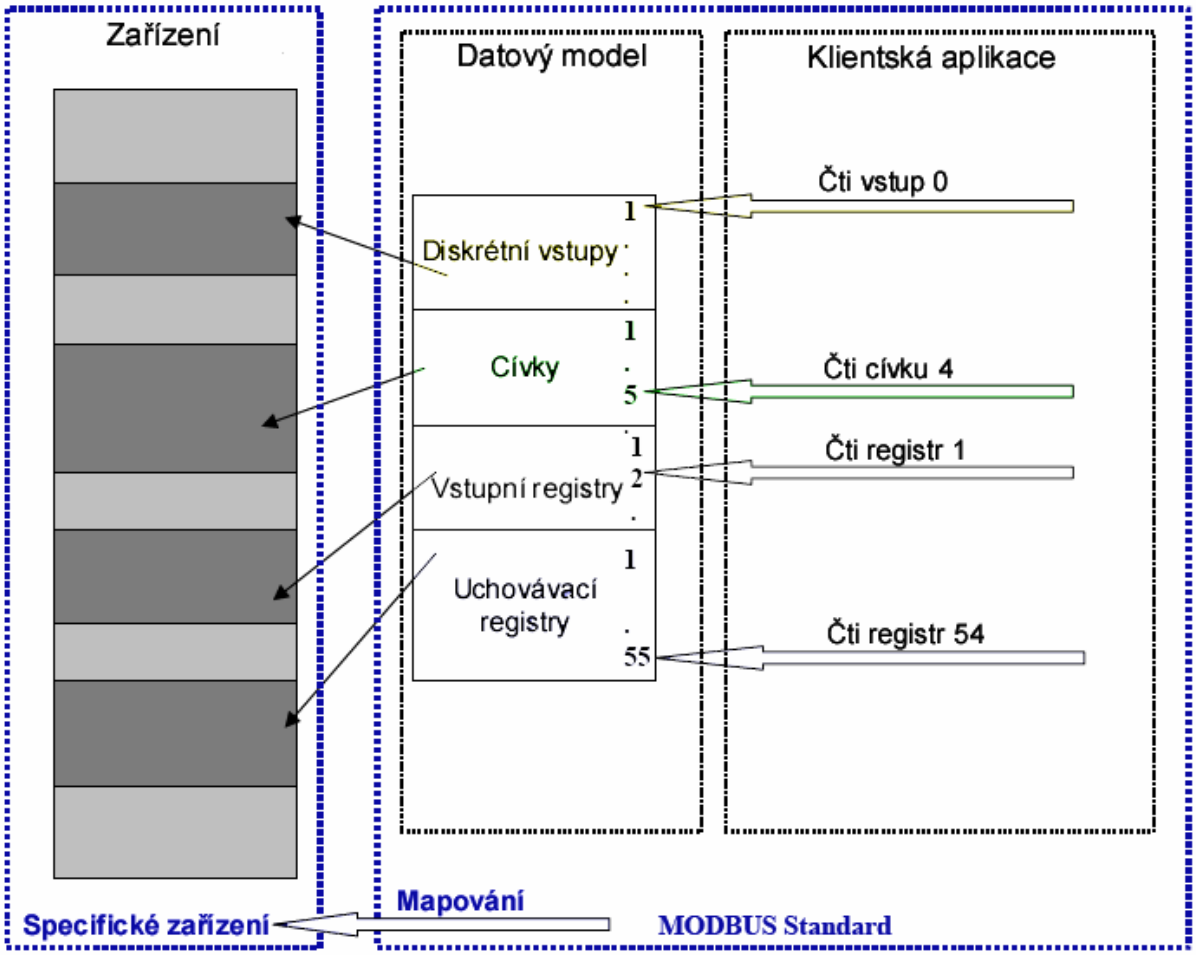
Standardem jsou nadefinovány chybové kódy, při výskytu neočekávané operace a tedy je určeno zpracování zprávy na straně hostitele. Tento cyklus je nastíněn na obrázku.



Obr. 2-3 Zpracování zprávy na straně hostitele [2]

2.2 Datový model

Každé hostitelské zařízení disponující komunikací pomocí Modbus protokolu má své specifické interní uspořádání dat v paměti do bloků, ke kterým přistupuje pomocí definovaných adres. Jelikož cizí zařízení bude tyto data využívat a neví, jak je v daném zařízení definován datový a adresní prostor, definuje standart Modbus datový model, ke kterému cizí zařízení přistupuje. Tento model je pak interně svázán na hostitelském zařízení s konkrétním datovým prostorem. Cizí zařízení tedy nemusí znát přesnou strukturu datového prostoru hostitelského zařízení.



Obr. 2-4 Datový model Modbus hostitele [1]

Cizí zařízení tedy může číst a zapisovat hodnoty vstupů a výstupů pomocí datového modelu. K jednotlivým prvkům přistupuje v datovém bloku pomocí 16 bitové adresy. Každý blok má rozsah 0..65 535. Pomocí jedné zprávy může klient operovat i s více prvky, avšak nesmí překročit omezení délky datového bloku zprávy.

Segmenty datového modelu již dle označení ukazují, jakým způsobem se s nimi bude zacházet. Některé jsou jen pro čtení, některé vrací 1 bit hodnotu, jiné 16 bit. Segmenty jsou navrženy tak, aby bylo možno ovládat všechny typy vstupů a výstupů, tedy digitální i analogové. Vše shrnuje následující tabulka.

	Typ	**Přístup**	**Popis**
Diskrétní vstupy (Discrete inputs)	1 bit	Pouze čtení	Data poskytovaná I/O systémem
Cívky (Coils)	1 bit	Čtení/zápis	Data modifikovatelná aplikačním programem

Vstupní registry (Input registers)	16 bit	Pouze čtení	Data poskytovaná I/O systémem
Uchovávací registry (Holding registers)	16 bit	Čtení/zápis	Data modifikovatelná aplikačním programem

Tab. 2-2 Segmenty datového modelu

2.3 Kódy funkcí

Každá žádost obsahuje kód funkce, kterou cizí aplikace požaduje po hostiteli, aby ji vykonal. Tyto kódy jsou jednak definované Modbus protokolem, také si každý uživatel, může definovat vlastní, potažmo firma může používat své definované kódy. Všechny kódy, které nejsou definované standardem, nemusí být daným hostitelským zařízením podporovány.

Všechny vlastnosti těchto tří kategorií funkcí shrnuje následující odstavec :

Veřejné kódy funkcí

- jasně a unikátně definované společností Modbus - IDA
- veřejně zdokumentované

Uživatelsky definované kódy funkcí

- dva rozsahy uživatelsky definovaných kódů funkcí: 65 .. 72 a 100 .. 110
- umožňují uživateli implementovat funkce, které nejsou definovány specifikací
- není garantována unikátnost kódů

Rezervované kódy funkcí

- kódy funkcí, které jsou v současnosti používány některými firmami
- nejsou dostupné pro veřejné použití

Všechny kódy funkcí definované standardem zobrazuje tabulka :

				Kódy funkcí			
				Kód	Podfunkce	hex	
Přístup k datům	**Bitový přístup**	Fyzické diskrétní vstupy	Čti diskrétní vstupy	**02**		**02**	
		Interní bity nebo fyzické cívky	Čti cívky	**01**		**01**	
			Zapiš jednu cívku	**05**		**05**	
			Zapiš více cívek	**15**		**0F**	
	16-bitový přístup	Fyzické vstupní registry	Čti vstupní registr	**04**		**04**	
		Interní registry nebo fyzické výstupní registry	Čti uchovávací registry	**03**		**03**	
			Zapiš jeden registr	**06**		**06**	
			Zapiš více registrů	**16**		**10**	
			Čti/zapiš více registrů	**23**		**17**	
			Zapiš registr s maskováním	**22**		**16**	
			Čti FIFO frontu	**24**		**18**	
	Přístup k záznamům v souborech		Čti záznam ze souboru	**20**	**6**	**14**	
			Zapiš záznam do souboru	**21**	**6**	**15**	
Diagnostika			Čti stav	**07**		**07**	
			Diagnostika	**08**	**00-18, 20**	**08**	
			Čti čítač kom. událostí	**11**		**0B**	
			Čti záznam kom. událostí	**12**		**0C**	
			Sděl identifikaci	**17**		**11**	
			Čti identifikaci zařízení	**43**	**14**	**2B**	
Ostatní			Zapouzdřený přenos	**43**	**13, 14**	**2B**	
			CANOpen základní odkaz	**43**	**13**	**2B**	

Tab. 2-3 Seznam kódů funkcí [2]

3 KNIHOVNA MODBUSLIBRARY

Vytvořené knihovny slouží pro komunikaci protokolem Modbus TCP. Jsou implementovány základní operace, které můžeme dle protokolu používat. Knihovna je logicky členěna na tři části : funkce pro obecnou práci s knihovnou, funkce pro ovládání klienta a funkce pro ovládání virtuálního serveru. Výstupem knihovny, tedy funkce a datové bloky ovládané uživatelem, jsou struktura klienta a serveru, dále funkce pro vytvoření nového klienta a serveru, funkce pro připojení klienta, funkce pro spuštění serveru a dílčí funkce na straně klienta vycházející z podporovaných operací daných standardem Modbus TCP.

Koncepce knihovny spočívá v rozšíření knihovny *winsock.h* o podporu protokolu Modbus. Je tedy její nadstavbou a zapouzdřuje ji tak, že uživatel používá pouze funkce vytvořené knihovny a o ostatní se nestará.

Jelikož je pro tuto práci stěžejní část, která má na starost ovládání klienta, rozebereme vnitřní strukturu a funkce klienta podrobněji. Strukturu a funkce virtuálního serveru pouze popíšeme, aby bylo jasné, na jakém principu virtuální server funguje. Nejdříve ale popíšeme funkce pro obecnou práci s knihovnou.

3.1 Popis funkcí knihovny

3.1.1 Obecné funkce

Funkce slouží pro inicializování knihovny, pro zobrazování a testování chyb, které nastaly při práci s knihovnou a pro její korektní ukončení. Také obsahuje funkce pro vrácení textového významu definovaného v knihovně.

Obsahuje funkce :

`int InitializeLibrary();` Funkce knihovnu zinicializuje, tedy zinicializuje knihovnu *winsock.h*
`int ReleaseLibrary();` Funkce knihovnu korektně ukončí, tedy ukončí práci s knihovnou *winsock.h*
`char* GetErrorMessage(WORD errorCode);` Funkce vrátí textovou zprávu s popisem chybového kódu

`int LastClientError(PClient client);`
`int LastServerError(PServer server);`
Funkce vrátí poslední chybu vzniklou s prací s knihovnou

Tab. 3-1 Obecné funkce knihovny

3.1.2 Funkce pro ovládání klienta

Klient je realizován jako struktura obsahující všechny potřebné informace pro klienta včetně jeho socketu. Uživatel ovládá klienta pomocí funkcí, kterým předá odkaz na tuto strukturu či jiné další potřebné informace.

Struktura klienta :

```
typedef struct SClient
{
     bool FConnected;
     char FIP[15];
     int FPort;
     unsigned char FLastError;
     WORD FMsgCounter;
     SOCKET FSocket;
     char FSendMessage[MSGSIZE];
     char FReadMessage[MSGSIZE];
     int FSendMessageSize;
     int FReadMessageSize;
}TClient, *PClient;
```

Proměnná FConnected signalizuje, zda je klient připojen. Proměnná FIP je textového typu a uchovává IP adresu hostitele. Proměnná FPort uchovává port, na který se má klient připojit. Proměnná FLastError obsahuje poslední chybu vzniklou při práci s klientem. Proměnná FMsgCounter obsahuje číslo Modbus zprávy, kterou může klient poslat, hodnota se automaticky zvyšuje a před přetečením se nuluje. Za jistých okolností může sloužit jako údaj o počtu poslaných zpráv klientem. Proměnná FSocket zastupuje obecný komunikační prostředek systému, pomocí kterého klient komunikuje se serverem. Poslední čtyři proměnné slouží pro uchování skutečně zaslaných a obdržených dat. Obsahují vlastní data a údaj o jejich délce.

Funkce pro ovládání klienta :

```
PClient NewClient(char *ip, int port);
```

Funkce vytvoří nového klienta (novou strukturu), nadefinuje základní hodnoty proměnných a vrátí ukazatel na nového klienta. Funkce vyžaduje jako parametr IP adresu a port serveru, kam se má připojit. Použití : `PClient mbClient = NewClient(“127.0.0.1”, 502);`

```
int DestroyClient(PClient client);
```

Funkce pro korektní ukončení práce s klientem. Ukončí spojení a uvolní klienta z paměti.

```
int ConnectClient(PClient client);
```

Funkce pro připojení klienta k běžícímu serveru. Zapouzdřuje funkce pro ovládání socketu, které jsou umístěny v knihovně *winsock.h*.

Pro vytvoření socketu se volá funkce :

```
SOCKET socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)
```

Po jejím skončení existuje socket pro komunikaci TCP/IP typu stream. Pro nasměrování roury socketu se naplní struktura `sockaddr_in` a s její pomocí se socket připojí na socket na serveru zavoláním funkce :

```
connect(SOCKET, (sockaddr*)&sockName, sizeof(sockName))
```

Po úspěšném skončení všech funkcí je klient připojen k serveru.

```
int DisconnectClient(PClient client);
```

Funkce, která odpojí připojeného klienta od serveru.

```
int SendMsg(PClient client, char *buffer, DWORD length);
```

Vnitřní funkce klienta, která pošle obecnou zprávu serveru pomocí protokolu TCP/IP. Zapouzdřuje funkci knihovny *winsock.h* :

```
send(SOCKET socket, const char *buf, int length, int flags)
```

Tato funkce zapouzdřuje funkce pro komunikaci po daném socketu a postará se o odeslání zprávy. Tuto zprávu uloží do proměnné `FSendMessage`.

```
int ReadMsg(PClient client, char *buffer, DWORD *read);
```

Vnitřní funkce klienta, která čte obecnou zprávu ze serveru pomocí protokolu TCP/IP. Zapouzdřuje funkci knihovny *winsock.h* :

```
recv(SOCKET socket, char *buf, int length, int flags)
```

Tato funkce zapouzdřuje funkce pro čekání na příchod zprávy, její přečtení, parsování, kontrolu správnosti zprávy a předá data zprávy pomocí parametru `buffer`. V našem (defaultním) případě pracuje v blokujícím režimu. To znamená, že se vlákno zastaví při vykonávání této funkce, dokud zpráva nedojde a nezpracuje se. Přečtenou zprávu uloží do proměnné `FReadMessage`.

```
int SendRequest(PClient client, unsigned char functionType,
WORD baseAddress, WORD count, unsigned char *data, unsigned
char dataLength);
```

Složí a odešle Modbus zprávu na server, pokud je klient připojen. Zapouzdřuje funkci `SendMsg`.

```
int GetAnswer(PClient client, WORD *MsgNo, unsigned char *DeviceId, unsigned char *functionCode, WORD *baseAddress, WORD count, unsigned char **data, unsigned char *dataLength);
```

Čeká na Modbus zprávu ze serveru, pokud je klient připojen. Zapouzdřuje funkci `ReadMsg`. Přijatou zprávu rozparsuje a vrací jednotlivá data.

```
int WriteCoil(PClient client, WORD number, bool stateon);
```

Výstupní funkce knihovny, která pošle zprávu serveru se žádostí o zapsání cívky.

```
int ReadCoil(PClient client, WORD number, bool *stateon);
```

Výstupní funkce knihovny, která pošle zprávu serveru se žádostí o vrácení stavu cívky.

```
int WriteHoldingRegister(PClient client, WORD number, WORD value);
```

Výstupní funkce knihovny, která pošle zprávu serveru se žádostí o zapsání uchovávacího registru.

```
int ReadHoldingRegister(PClient client, WORD number, WORD *value);
```

Výstupní funkce knihovny, která pošle zprávu serveru se žádostí o vrácení stavu uchovávacího registru.

```
int ReadCoils(PClient client, WORD from, WORD count, bool **stateons);
```

Výstupní funkce knihovny, která pošle zprávu serveru se žádostí o vrácení stavů více cívek.

```
int ReadHoldingRegisters(PClient client, WORD from, WORD count, WORD **values);
```

Výstupní funkce knihovny, která pošle zprávu serveru se žádostí o vrácení stavů více uchovávacích registrů.

```
int ReadDiscreteInputs(PClient client, WORD from, WORD count, bool **stateons);
```

Výstupní funkce knihovny, která pošle zprávu serveru se žádostí o vrácení stavů více diskrétních vstupů.

```
int ReadInputRegisters(PClient client, WORD from, WORD count, WORD **values);
```

Výstupní funkce knihovny, která pošle zprávu serveru se žádostí o vrácení stavů více vstupních registrů.

```
int WriteCoils(PClient client, WORD from, WORD count, bool stateon[]);
```

Výstupní funkce knihovny, která pošle zprávu serveru se žádostí o zapsání více cívek.

```
int WriteHoldingRegisters(PClient client, WORD from, WORD count, WORD values[]);
```

Výstupní funkce knihovny, která pošle zprávu serveru se žádostí o zapsání více uchovávacích registrů.

```
int GetRWBuffer(PClient client, char **RBuffer, char **WBuffer, int *RBufferSize, int *WBufferSize);
```

Funkce vrátí obsah proměnných `FReadMessage` a `FSendMessage` jako ukazatele na pole znaků a vrátí jejich délky. Proměnné obsahují surová data, která se poslala při vyslání požadavku, a přijala jako odpověď na tento požadavek.

Většina funkcí vyžaduje parametry *from* a *count*. Parametr *from* udává vždy počáteční prvek souboru prvků, nad kterým má server vykonávat dané operace. Parametr *count* udává počet prvků tohoto souboru.

3.1.3 Funkce pro ovládání serveru

Server je koncipován jako samostatný modul běžící ve vlastním vlákně, který po vytvoření naslouchá na localhost portu. Po připojení klienta ho obslouží dle standardu Modbus. Podporuje funkce pro zapisování a čtení nad datovým modelem Modbus standardu.

Programově je server koncipován jako struktura, která si uchovává všechny potřebné informace a stavy, na kterou je dále navázána struktura s datovým modelem.

Server běží samostatně a je nutno ho jen vytvořit a ukončit.

```
PServer NewServer(int port, WORD serverMaxClients, WORD nDiscreteInputs, WORD nCoils, WORD nInputRegisters, WORD nHoldingRegisters);
```

Funkce vytvoří nový server a vrátí jeho ukazatel. Obvykle Modbus server poslouchá na portu 502 dle standardu. Parametr serverMaxClients definuje maximální počet čekajících klientů na spojení se serverem. Další čtyři parametry definují počty prvků v datových segmentech datového prostoru serveru, kterým server disponuje a které jsou v této funkci nainicializovány. Použití :

```
PServer mbServer = NewServer(502, 10, 16, 16, 8, 8);
```

```
int DestroyServer(PServer server);
```

Funkce ukončí činnost serveru a korektně ho uvolní z paměti

```
int EstablishServer(PServer server);
```

Funkce nainicializuje server a spustí ho. Server bude ve stavu naslouchání. Server běží ve svém vlastním vlákně, takže po spuštění serveru funkce skončí.

```
int ShutdownServer(PServer server);
```

Funkce ukončí činnost serveru. Server stále existuje, takže po zavolání funkce `EstablishServer` bude server znovu fungovat.

```
int ChangeData(PServer server, bool discreteInputs[], WORD
inputRegisters[], bool **coils, WORD **holdingRegisters);
```

Funkce slouží pro manipulaci s daty na virtuálním serveru. Pomocí parametrů `discreteInputs` a `inputRegisters` se zapisují příslušné hodnoty do datové oblast na serveru. Pomocí parametrů `coils` a `holdingRegisters` jsou vrácena data ze serveru. Funkce slouží pro ovládání dat na virtuálním serveru z ovládací aplikace.

3.2 Realizace knihovny

Knihovnu tvoří jediný soubor, a to *ModbusLibrary.dll*. Jedná se tedy o dynamicky linkovanou knihovnu pro operační systém Windows. Pro vytvoření knihovny je nutno kompileru sdělit, že chceme vytvořit DLL knihovnu a nadefinovat vstupní bod. Vstupní bod je funkce, která je volána při načtení a uvolnění knihovny. V této funkci můžeme vykonat všechny potřebné operace pro inicializaci a korektní ukončení práce s knihovnou.

Knihovna je psána v C/C++ unmanaged kódu a zkompilována se standardní __cdecl konvencí volání funkcí, tedy parametry funkcí se předávají zásobníkem v pořadí zprava doleva a o vyčištění zásobníku se stará volající. Tvorba v unmanaged kódu dovoluje přímý přístup k paměti, díky kterému ale vzrůstá pravděpodobnost chyb práce s pamětí a k neoprávněnému přístupu do paměti. Můžeme také používat přímo API funkce, kterými operační systém disponuje. Aplikace vytvořené v unmanaged kódu jsou také

rychlejší, protože se nevykonávají různé ochranné mechanismy, a jsou překládány přímo do strojového kódu, tedy nepotřebují pro svůj běh žádné další runtime prostředí.

Potřebné funkce pro práci s knihovnou jsou označeny pomocí direktivity `__declspec(dllexport)`, a jelikož překladač dává funkcím jména i s údajem parametrů kvůli přetěžování je použita ještě direktiva `extern „C“`. Jména funkcí v DLL jsou potom tvořena čistě jen jmény prototypů funkcí. Druhou variantou je použít externí soubor *ModbusLibrary.def* (module definition file), ve kterém nadefinujeme jména funkcí v závislosti na ordinální hodnotě v knihovně. Toto ovšem vyžaduje jeho nadefinování při linkování. Tato metoda ale počítá s tím, že se již knihovna nebude měnit, zejména se nebudou odebírat ani přidávat další exportované funkce.

3.3 Použití knihovny

Způsob použití knihovny závisí na programovacím prostředí, ve kterém budeme knihovnu používat. Vždy je ale nutno znát cestu umístění souboru knihovny nebo ho nejlépe umístit do adresáře cílové aplikace.

3.3.1 Programovací prostředí

Programovací prostředí se v zásadě dělí na managed a unmanaged kód. To znamená, zda budeme využívat rozhraní CLR, či přistupovat k systémovým funkcím pomocí API funkcí.

Unamanaged kód je přímo překládán do strojového jazyka. Využívá API funkce, které jsou podporovány operačním systémem a standardní knihovny. Přístup k paměti je přímý bez ochranných mechanismů, a proto se může stát, že dojde k neoprávněnému přístupu, či nastane jiná chyba paměti.

Managed kód byl vyvinut s požadavkem na nezávislost na platformě. Využívá rozhraní CLR (Common Language Runtime), které poskytují knihovny distribuované balíčkem .Net Framework. Managed kód je překladačem přeložen do IL (popř. MSIL) kódu. Tato aplikace musí proto být spouštěna pomocí runtime prostředí, které převede kód IL do strojového kódu. Toto prostředí bylo zavedeno také s požadavkem na bezpečnost. Proto disponuje bezpečnostními mechanismy, které zabezpečují korektní práci s pamětí a se zdroji. Prostředí také disponuje určitým komfortem. Garbage Collector je nástroj, který se stará o korektní ukončení práce s proměnnými a objekty. Toto prostředí představuje nový způsob programování, které je bezpečné a platformově nezávislé.

3.3.2 Prostředí unmanaged kódu

V programech psaných v unmanaged C/C++ kódu by se použila pro načtení knihovny API funkce `LoadLibrary()` a pro získání ukazatele na požadovanou funkci API funkce `GetProcAddress()`. Samozřejmě musíme znát parametry funkcí, nejlépe z hlavičkového souboru. Tímto způsobem lze s knihovnou pracovat. Po skončení práce je třeba zase knihovnu uvolnit. Situace je zachycena na demonstrativním příkladě, kdy se zavolá funkce *NewClient* :

```
#include "stdio.h"
#include "windows.h"
typedef HWND (WINAPI* NEWCLIENT)(char*, int);
HINSTANCE library;

int main(int argc, char* argv[])
{
	NEWCLIENT NewClient = NULL;
	if((library = LoadLibrary(L"ModbusLibrary.dll")) == NULL)
		return GetLastError();
	NewClient = (NEWCLIENT)GetProcAddress(library, "NewClient");
	if(NewClient != NULL)
		HWND client = NewClient("127.0.0.1", 502);
	free(client);
	FreeLibrary(library);
	return 0;
}
```

Druhým a jednodušším způsobem by bylo použít zdrojové soubory knihovny a přilinkovat je. V aplikaci by tedy bylo možné okamžitě používat funkce, kterými knihovna disponuje. Uživatel dokonce přesně zná, jak funkce přesně pracují, a může tedy přizpůsobit jejich obsluhu. To ovšem vyžaduje, aby měl cílový uživatel k dispozici kompletní zdrojové kódy knihovny.

3.3.3 Prostředí managed kódu

V prostředí managed kódu C++, tedy v prostředí používající rozhraní CLR (Common Language Runtime) můžeme získat prototypy funkcí z hlavičkového souboru a to jeho přilinkováním. V tomto případě musíme také přilinkovat reference jmen funkcí pro linker. Tedy musíme přilinkovat statickou knihovnu, která je pro tyto účely vytvářena aplikací Microsoft Visual C++ spolu s dynamickou knihovnou. Statická knihovna

neobsahuje exekutivní části funkcí, ale je to jen tabulka jmen funkcí a adres pro linker. Situace je znázorněna na demonstrativním příkladu :

```
#include "ModbusLibrary.h"
#pragma comment(lib, "ModbusLibrary.lib")
namespace priklad {
	...
	public ref class Form1 : public System::Windows::Forms::Form
	{
	...
		private: System::Void button1_Click(...) {
			System::IntPtr client = System::IntPtr::Zero;
			client = (System::IntPtr)NewClient("127.0.0.1", 502);
		}
	};
}
```

3.3.4 Prostředí C#

Většinou se ale knihovna bude používat v jiných jazykových prostředích. Proto je nutné v každém prostředí propojit jména funkcí z DLL knihovny s konkrétními prototypy funkcí.

Při použití v managed prostředí jazyka C#, ve kterém budeme tvořit ovládací grafickou aplikaci, je nutné definovat prototypy funkcí v managed kódu. Proto je zavedena sada funkcí *PInvoke* [3], která se o převod postará automaticky sama a volá se použitím funkce *DllImport*. Aby příkaz fungoval musí být přilinkovaná nadřazená knihovna *InteropServices* pomocí příkazu preprocesoru : `using System.Runtime.InteropServices`.

Funkce *PInvoke* poskytne funkci automaticky právo (Security.Permission) k vykonávání unmanaged kódu, tedy nezabezpečný přístup k paměti a zdrojům. Tímto se zabezpečí, že nebudou aktivní ochranné mechanismy managed kódu, které by bránily funkcím z knihovny ve vykonávání operací.

Funkci je potom již možné využívat v prostředí managed kódu s konvencí volání __clrcall. Datové typy je nutno logicky přepsat na objektové typy dle tabulky převodu datových typů. U datových typů, kde není jednoznačně určeno příslušnost k managed typu, resp. si neodpovídají konvencí, kódováním či realizací v paměti, se musí tyto typy přenést z managed contextu do unmanaged contextu či naopak pomocí třídy *Marshal* nebo přímo v deklaraci prototypu funkce direktivou *MarshalAs* a nadefinováním požadovaného cílového typu.

Použití knihovny v prostředí C# demonstruje následující příklad :

Soubor Library.cs

```
using System.Runtime.InteropServices;
namespace Library
{
  public class Client
  {
    [DllImport("ModbusLibrary.dll", EntryPoint = "NewClient", CharSet
= CharSet.Unicode)] public static extern IntPtr NewClient([
MarshalAs(UnmanagedType.LPStr)]String IP, UInt16 Port);

    [DllImport("ModbusLibrary.dll", EntryPoint = "ConnectClient",
CharSet = CharSet.Unicode)] protected static extern Int32
ConnectClient(IntPtr Client);

    [DllImport("ModbusLibrary.dll", EntryPoint = "DestroyClient",
CharSet = CharSet.Unicode)] public static extern Int32
DestroyClient(IntPtr Client);

    [DllImport("ModbusLibrary.dll", EntryPoint = "ReadCoil", CharSet =
    CharSet.Unicode)] public static unsafe extern Int32
ReadCoil(IntPtr client, UInt16 number, Boolean* stateon);
  }
}
```

Soubor Form1.cs

```
using Library;
namespace Priklad
{
  public partial class Form1 : Form
  {
    ...
    IntPtr client = IntPtr.Zero;
    ...
    private void button1_Click(object sender, EventArgs e)
    {
      client = Client.NewClient("127.0.0.1", 502);
      Client.ConnectClient(client);
      Boolean tmpState = false;
```

```
      unsafe
      {
        DllReadCoil(client, 1, &tmpState);
      }
      CheckBox1.Checked = tmpState;
      Client.DestroyClient(client);
    }
  };
}
```

3.3.5 Rozhraní knihovny v prostředí C#

Pro účely vytvoření vizualizační řídící aplikaci vytvoříme rozhraní pro práci s knihovnou v prostředí jazyka C#. Vytvoříme tedy soubor, který se bude distribuovat spolu s DLL knihovnou a který se bude vkládat do projektu cílové aplikace.

3.3.5.1 Popis rozhraní

Vytvořený soubor rozhraní obsahuje prostor jmen *ModbusLibrary*, který zastřešuje vytvořené třídy. Vytvořili jsme celkem tři třídy. První pro funkce, které obecně pracují s knihovnou, a pro nadefinování chybových kódů. Tato třída – *Modbus* – je abstraktní a nemůže se dědit. Výstupem druhé třídy – *ModbusClient* – jsou funkce pro práci s klientem. Celá tato třída je koncipována jako objekt klienta, se kterým je pracováno. Třetí třída – *ModbusServer* – je koncipována jako objekt serveru.

Aplikace vytvořené v tomto prostředí sice obsahují GarbageCollector, ale musíme si uvědomit, že vyvtořené třídy pracují s pamětí na nižší úrovní a to v dll knihovně. Proto musíme dodržet sled a úplnost podružných funkcí.

3.3.5.2 Práce s funkcemi vyžadující ukazatele

Zastřešením funkcí z DLL knihovny odpadají v cílovém projektu starosti s parametry funkcí a zejména práce s pointery, které představují u některých funkcí jejich výstupy. Proto musíme ve vytvořené třídě použít u prototypu exportované funkce ještě direktivitu unsafe, která zajistí, že můžeme pracovat s pointery v prostředí unsafe. V nastavení compilátoru musí být povolena kompilace unsafe kódu, tedy musí být v příkazovém řádku pro kompilaci obsažen atribut `/unsafe`. Tato funkce definovaná v unsafe kontextu musí být také z unsafe kontextu volána. Tyto proměnné mohou být deklarovány i v managed kontextu, neboť se automaticky přenesou přes hranici

contextů. Naopak to ale automaticky neplatí a musí být použita sada funkcí třídy Marshal pro přenesení přes hranici kontextu.

3.3.5.3 Ošetření neočekávaných stavů - EventHandler

Třídy klienta a serveru používají funkce z DLL knihovny, přičemž může vzniknout mnoho očekávaných i neočekávaných vyjímek. Tyto vyjímky jsou odchytávány a uživatel je o nich informován prostřednictvím událostí. Každá vyjímka vyvolá událost, kterou zpracuje *EventHandler* a předá ji nadefinované funkci uživatele. *EventHandler* je objekt sloužící pro zachytávání daných událostí a jejich zpracování či předání cílové výkonné funkci. Pomocí námi vytvořené třídy *ErrorEventArgs* můžeme předat funkci parametry obsahující chybový kód a textovou zprávu. Je ale nutno parametr *EventArgs* ve volané funkci přetypovat na proměnnou *ErrorEventArgs*, která je jejím potomkem. Potom může volaná funkce tuto zprávu zpracovat.

3.3.5.4 Strutktura rozhraní

V novém prostoru jmen *ModbusLibrary* jsme vytvořili třídy *ErrorEventArgs*, *Modbus*, *ModbusClient* a *ModbusServer*. Jejich rozhraní z pohledu uživatele je zachyceno na diagramech, tedy jsou zde zachyceny všechny public funkce, metody a proměnné.

class ErrorEventArgs : EventArgs
property UInt16 *ErrorCode* property String *Message*
constuctor *ErrorEventArgs*(UInt16 ErrorCode, String Message)

class Modbus
enum *Chyby*: byte
String *GetErrorMessage*(UInt16 ErrorCode) UInt16 *LastServerError*(IntPtr Server) UInt16 *LastClientError*(IntPtr Client)

class ModbusClient
event EventHandler *OnError* property bool *EventInvalidCreation*
constructor *ModbusClient*(String IP, UInt16 Port) constructor *ModbusClient*() void *DisconnectClient*() void *ConnectClient*() void *WriteCoil*(UInt16 Number, Boolean StateOn) void *ReadCoil*(UInt16 Number, ref Boolean StateOn) void *WriteHoldingRegister*(UInt16 Number, UInt16 Value) void *ReadHoldingRegister*(UInt16 Number, ref UInt16 Value) void *ReadCoils*(UInt16 From, UInt16 Count, ref Boolean[] StateOns) void *ReadHoldingRegisters*(UInt16 From, UInt16 Count, ref UInt16[] Values) void *ReadDiscreteInputs*(UInt16 From, UInt16 Count, ref Boolean[]

StateOns) void ReadInputRegisters(UInt16 From, UInt16 Count, ref UInt16[] Values) void WriteCoils(UInt16 From, UInt16 Count, Boolean[] StateOns) void WriteHoldingRegisters(UInt16 From, UInt16 Count, UInt16[] Values) Boolean RefreshMomentum(Boolean[] DigitalOutputs, UInt16[] AnalogOutputs, ref Boolean[] DigitalInputs, ref UInt16[] AnalogInputs) void GetRWBuffer(out String SendMessage, out String ReadMessage)

class ModbusServer
event EventHandler *OnError* property bool *EventInvalidCreation*
constructor *ModbusServer*(UInt16 Port, UInt16 MaxClients, UInt16 nDiscreteInputs, UInt16 nCoils, UInt16 nInputRegisters, UInt16 nHoldingRegisters) constructor *ModbusServer*() void *ShutdownServer*() void *EstablishServer*() void *ChangeData*(ref Boolean[] Coils, Boolean[] DiscreteInputs, ref UInt16[] HoldingRegisters, UInt16[] InputRegisters)

Tab. 3-1 Diagramy výstupního interface rozhraní DLL pro C#

3.4 Shrnutí

Vytvořili jsme knihovnu *ModbusLibrary.dll* v jazyce C pro implementaci Modbus TCP protokolu přes TCP/IP komunikaci. Knihovna obsahuje funkce jak pro ovládání klienta, tak pro vytvoření a spuštění virtuálního serveru. Vylepšením knihovny by bylo ošetření všech možných chybových stavů vzniklých jak při přímé práci s paměti, tak při práci se sockety. To ale vyžaduje dlouhodobé testování a debuggování.

Dále jsme naprogramovali třídy v jazyce C# pro efektivní práci s knihovnou ve vizualizační uživatelské aplikaci. Výstupem jsou tedy třída klienta, třída serveru, obecná abstraktní třída a třída pro chybový parametr. Třídy zastřešují použití DLL knihovny, vytvořené v jiném prostředí. Pro cílový projekt je přilinkováním souboru s třídami výrazně zjednodušena práce s knihovnou. Možným vylepšením tříd je použití fronty uživatelem požadovaných funkcí, vykonáním těchto funkcí ve zvláštním vlákně a po skončení vyvoláním události.

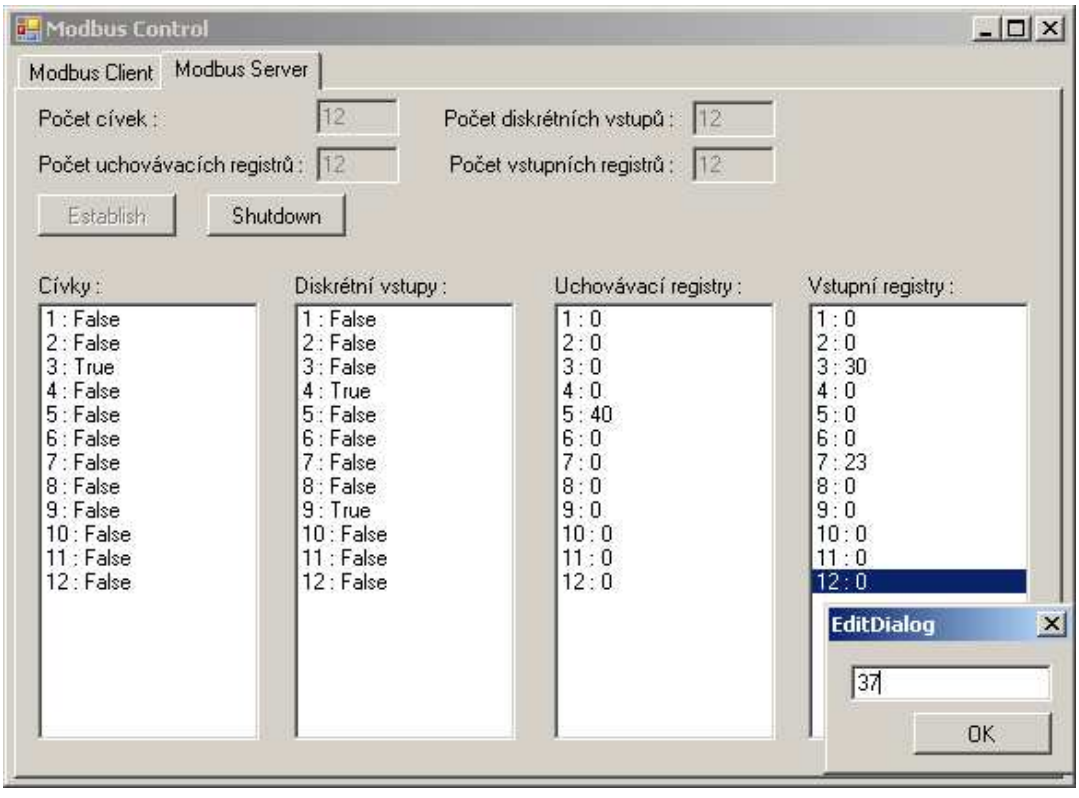
Soubory ModbusLibrary.h, ModbusLibrary.dll a ModbusLibrary.cs (soubor s třídami pro použití v prostředí C#) jsou přiloženy.

4 MODBUS APLIKACE

Za účelem ukázat možnosti využití naprogramované knihovny *ModbusLibrary.dll* a rozhraní pro prostředí C# *ModbusLibrary.cs* jsme se rozhodli vytvořit další GUI aplikaci. Tato aplikace bude sloužit čistě k demonstrativním účelům. Aplikace se skládá z klienta a z virtuálního serveru. Každá část je ve své vlastní kartě a její ovládání intuitivní.

4.1 Server

Virtuální server pracuje samostatně a běží na počítači, kde je spuštěn. Po jeho spuštění se může připojit klient a vysílat své požadavky. Na kartě serveru, kde se server spouští a ukončuje, může uživatel libovolně měnit stavy diskrétních vstupů a hodnoty vstupních registrů, a to poklepáním na daný údaj. Aktualizace dat v zobrazovacích polích na straně serveru probíhá při přepnutí karet, takže při manipulaci s daty jsou data na serveru a v GUI rozhraní synchronní. Serverová část využívá vytvořené rozhraní a zejména funkce pro vytvoření a ukončení serveru a pro výměnu dat mezi datovou oblastí na serveru a GUI rozhraním.

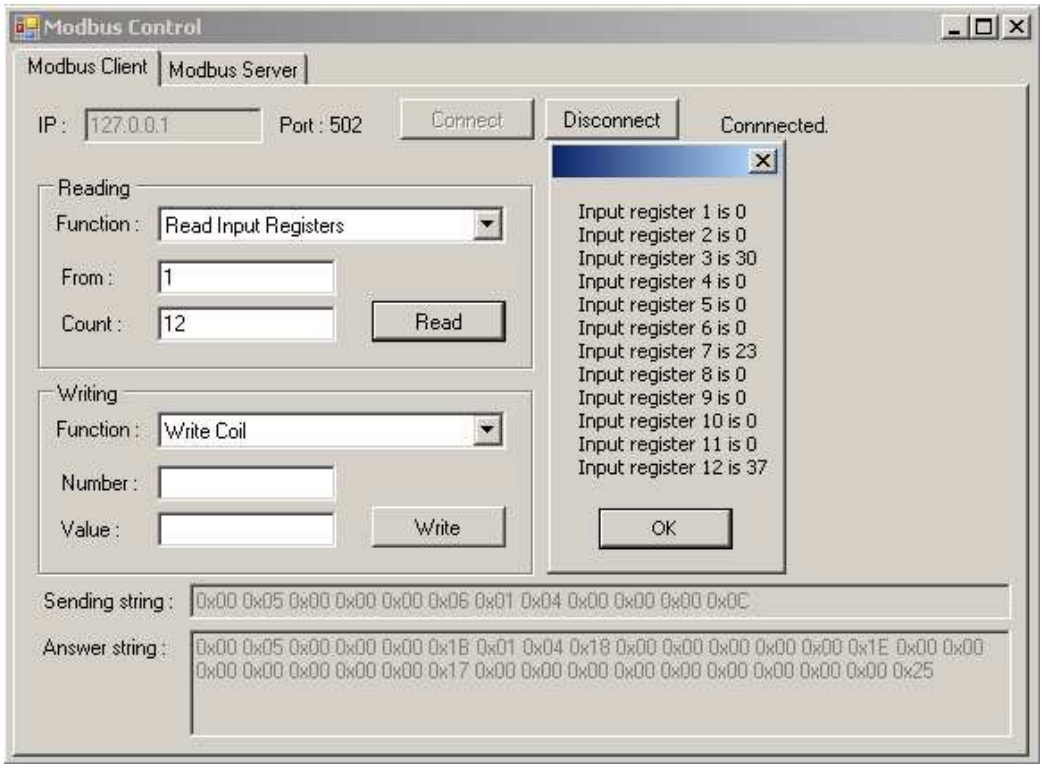


Obr. 4-1 Screenshot serverové části aplikace Modbus Control

4.2 Klient

Na kartě klienta uživatel může prostřednictvím ovládacích políček posílat příkazy na server a využívat tak nabízené funkce podporující Modbus protokol. Po odeslání příkazu pro čtení tlačítkem Read se zobrazí hláška s odpovědí, která obsahuje požadovaná data nebo chybovou hlášku. Po odeslání žádosti o zápis se zobrazí hláška o úspěšném zapsání nebo s chybovou hláškou. Při každé operaci čtení a zápisu se zobrazí skutečně poslaná data v políčkách nacházejících se ve spodní části. V horním políčku se zobrazí odeslaný paket se žádostí a ve spodním se zobrazí přijatý paket s odpovědí.

Klient podporuje tyto funkce pro čtení : čtení jedné cívky, čtení jednoho uchovávacího registru, čtení cívek, čtení uchovávacích registrů, čtení vstupních registrů a čtení diskrétních vstupů. Pro zápis jsou k dispozici tyto funkce : zapiš cívku a zapiš uchovávací registr.



Obr. 4-2 Screenshot části klienta aplikace Modbus Control

5 MODICON TSX MOMENTUM

Jedná se o zařízení firmy Modicon (Telemecanique - Schneider), které je koncipováno jako Modbus server. Tedy vykonává požadavky klientů na fyzickém zařízení. Tento převodník mezi Modbus příkazy a procesní instrumentací se skládá ze tří částí. Z vlastního procesoru s operačním programem, z komunikačního adaptéru pro ethernet (typ 170 ENT 110 00) a z vstupně/výstupní periferní karty (typ 170 ANR 120 90). Zařízení je tedy rozhraní mezi Modbus TCP komunikací a konkrétní procesní instrumentací. Komunikuje prostřednictvím ethernetu, tedy je lokalizovatelné v síti svojí IP adresou. Je vybaveno pro připojení k ethernetové síti standardním konektorem RJ-45.

5.1 Modbus podpora

Zařízení podporuje pouze dva Modbus příkazy, a to zápis a čtení uchovávacích (interních) registrů. Parametry těchto příkazů jsou definované výrobcem vstupně/výstupní základny a obsahují stavy digitálních i analogových vstupů a výstupů. Parametry obou příkazů jsou ve formě 12 slov a jsou dále popsány.

5.1.1 Ovládání vstupů

Při použití příkazu a čtení uchovávacích registrů se přečte 12 registrů (typu word), jejichž význam shrnuje následující tabulka :

1. word	Stav zařízení
2. word	Stav 8 diskrétních vstupů
3. word	Hodnota analogového vstupního kanálu č.1
4. word	Hodnota analogového vstupního kanálu č.2
5. word	Hodnota analogového vstupního kanálu č.3
6. word	Hodnota analogového vstupního kanálu č.4
7. word	Hodnota analogového vstupního kanálu č.5
8. word	Hodnota analogového vstupního kanálu č.6
9. .. 12. word	nepoužito

Tab. 5-1 Význam vstupního registru

5.1.1.1 Stav zařízení (1. word)

Hodnota obsahuje informace o stavu zařízení a o stavu 8 diskrétních výstupů.

Bity 15 .. 9	nepoužito
Bit 8	0 .. chyba modulu, 1 .. modul v pořádku
Bity 7 .. 4	nepoužito
Bit 3	stav kanálu 7 a 8 : 0 .. chyba, 1 .. v pořádku
Bit 2	stav kanálu 5 a 6 : 0 .. chyba, 1 .. v pořádku
Bit 1	stav kanálu 4 a 3 : 0 .. chyba, 1 .. v pořádku
Bit 0	stav kanálu 2 a 1 : 0 .. chyba, 1 .. v pořádku

Tab. 5-2 Význam bitů v registru stav zařízení

5.1.1.2 Stav diskrétních vstupů (2. word)

Hodnota obsahuje vpravo zarovnaný údaj o stavu diskrétních vstupů. Každý bit reprezentuje jeden vstup. Hodnota 0 je pro stav OFF a hodnota 1 je pro stav ON. Mapování bitů na vstupy je zachyceno na obrázku č. 5–1.

5.1.1.3 Hodnota analogových vstupů (3. – 8. word)

Každá hodnota obsahuje výsledek A/D převodu daného analogového vstupu. 15. a 0. bit je vždy nulový, to znamená, že rozsah je omezen na 0 .. 32766 (0x7FFE) a že rozlišení je zmenšeno na polovinu. Tento rozsah reprezentuje rozsah A/D převodníku 0 .. 10 V. Přičemž přesná reprezentace hodnot dle výrobce je zaznamenána v tabulce :

Vstupní napětí	**Výsledek A/D převodu**
10,000 V	32 000
10,238 V	32 760
> 10,238 V	32 766 (chyba přetečení)

Tab. 5-3 Mezní hodnoty A/D převodníku

5.1.2 Ovládání výstupů

Při použití příkazu a zápisu uchovávacích registrů se zapíše 12 registrů (typu word), jejichž význam shrnuje následující tabulka :

1. word	Systémové informace
2. word	Registr pro reakci diskrétních výstupů v nouzovém stavu
3. word	Registr pro reakci analogových výstupů v nouzovém stavu
4. word	Uživatelem definovaná hodnota analogového výstupu č.1 v nouzovém stavu
5. word	Uživatelem definovaná hodnota analogového výstupu č.2 v nouzovém stavu
6. word	Uživatelem definovaná hodnota analogového výstupu č.3 v nouzovém stavu

7. word	Uživatelem definovaná hodnota analogového výstupu č.4 v nouzovém stavu
8. word	Stav 8 diskrétních výstupů
9. word	Hodnota analogového výstupu č.1
10. word	Hodnota analogového výstupu č.2
11. word	Hodnota analogového výstupu č.3
12. word	Hodnota analogového výstupu č.4

Tab. 5-4 Význam výstupního registru

5.1.2.1 Systémové informace (1. word)

Hodnota musí být vždy větší než nula. „Nula se používá pro restart přístroje a jejím zápisem do registru může způsobit jeho poškození.“ [6]

Nastavením bitu 15 se povoluje nastavení hodnot výstupů v základním stavu pomocí registrů pro reakci v nouzovém stavu. V opačném případě se automaticky resetují. Pro setrvání výstupů v aktuální hodnotě bude potřeba nastavit tento bit.

5.1.2.2 Reakce diskrétních výstupů v nouzovém stavu (2. word)

Registr kombinuje uživatelsky definované hodnoty a nastavení reakce v nouzovém stavu pro diskrétní výstupy.

Pro setrvání diskrétních výstupů v aktuální hodnotě je nutno nastavit bit 15 a bit 14 ponechat nenastaven. Pro nastavení uživatelsky definovaných hodnot v normálním stavu je nutno nastavit i bit 14 a tyto hodnoty nadefinovat v bitech 0 .. 7.

Bity 0. . 7	Definované hodnoty diskrétních výstupů 1 .. 8 v nouzovém stavu
Bity 8 .. 13	Nepoužito
Bit 14	0 .. držet poslední stav, 1 .. nastavit definované hodnoty
Bit 15	0 .. resetovat všechny výstupy, 1 .. dle bitu 14

Tab. 5-5 Význam bitů v registru 2

5.1.2.3 Reakce analogových výstupů v nouzovém stavu (3. word)

Registr obsahuje nastavení reakce v nouzovém stavu pro analogové výstupy.

Bity 0. . 1	Nastavení 1. analogového výstupu
Bity 2 .. 3	Nastavení 2. analogového výstupu
Bity 4 .. 5	Nastavení 3. analogového výstupu
Bity 6 .. 7	Nastavení 4. analogového výstupu
Bity 8 .. 15	Nepoužito

Tab. 5-6 Význam bitů v registru 3

Možné hodnoty stavů pro každý kanál :

00	Minimální hodnota výstupu
01	Držet poslední hodnotu
10	Nastavit definovanou hodnotu
11	Držet poslední hodnotu

Tab. 5-7 Hodnoty nastavení nouzového stavu

5.1.2.4 Definované hodnoty výstupů v nouzovém stavu (4. – 7. word)

Registry obsahuji uživatelem definované hodnoty, které se mají nastavit na výstupech v případě nouzového stavu. Hodnota musí být platná, tedy v rozmezí 0 .. 32766 (0x7FFE).

5.1.2.5 Stav diskrétních výstupů (8. word)

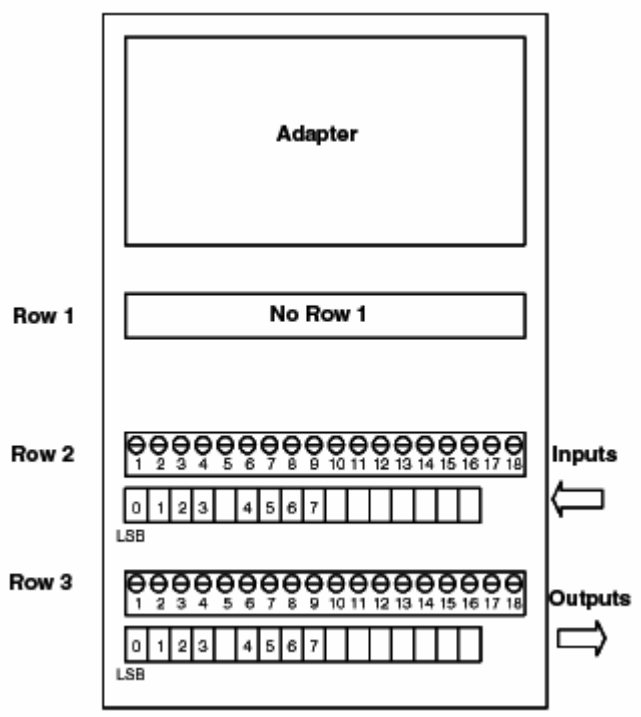
Hodnota musí obsahovat vpravo zarovnaný údaj o cílovém stavu diskrétních výstupů. Každý bit reprezentuje jeden výstup. Hodnota 0 je pro stav OFF a hodnota 1 je pro stav ON. Mapování bitů na výstupy je zachyceno na obrázku č. 5-1.

5.1.2.6 Hodnota analogových výstupů (9. – 12. word)

Každá hodnota obsahuje výsledek D/A převodu daného analogového výstupu. 15. a 0. bit je vždy nulový, to znamená, že rozsah je omezen na 0 .. 32766 (0x7FFE) a že rozlišení je zmenšeno na polovinu. Tento rozsah reprezentuje rozsah D/A převodníku 0 .. 10 V. Přičemž přesná reprezentace hodnot dle výrobce je zaznamenána v tabulce :

Výsledek D/A převodu	**Vstupní napětí**
32 000	10,000 V
32 760	> 10,238 V
32 766	10,238 V (chyba přetečení)

Tab. 5-8 Mezní hodnoty D/A převodníku



Obr. 5-1 Mapování diskrétních vstupů a výstupů na sběrnici [5]

5.2 Způsob ovládání

Rozhraní se ovládá pomocí výše uvedených Modbus požadavků se specifickými parametry. Zařízení pracuje v režimu, kdy po přijetí požadavku o zápisu uchovávacích registrů tuto operaci vykoná a setrvá v tomto stavu (stav ready) jednu sekundu. Poté se resetuje, tedy všechny výstupy jsou neaktivní ve stavu vypnuto. Zařízení se nachází v základním stavu. Pro online práci se zařízením je tedy nutné komunikovat se zařízením periodicky s intervalem menším než jedna sekunda. To je výhodné v případech, kdy potřebujeme komunikovat se zařízením v krátkých časových intervalech z důvodu real-timové korespondence vstupů a výstupů.

Druhým způsobem, kdy nepotřebujeme často komunikovat využijeme nastavení jednotky pomocí registru *Systémové informace* a registrů *Reakce v nouzovém stavu*, kdy nadefinujeme, že zařízení nemá automaticky resetovat výstupy, ale držet poslední zapsanou hodnotu, a tedy můžeme komunikovat s jednotkou, kdy je to potřeba.

Data se s jednotkou vyměňují prostřednictvím požadavků o čtení a zápisu uchovávacích registrů. Z vnitřního nastavení zařízení vyplývá, že se do těchto registrů při požadavku o zápisu načtou stavy vstupů a z registrů se nastaví stavy výstupů. Je tedy zřejmé, že pokud budeme chtít získat stavy vstupů, musíme nejdříve vyslat požadavek o zápisu registrů, a tím i jejich aktualizace dle fyzického stavu vstupně výstupní základny. Poté můžeme zaslat požadavek o vrácení stavu registrů. Použitím jiného způsobu ovládání hrozí nebezpečí, že získaná data nebudou odpovídat fyzickému stavu zařízení.

5.2.1 Rozšíření rozhraní v prostředí C#

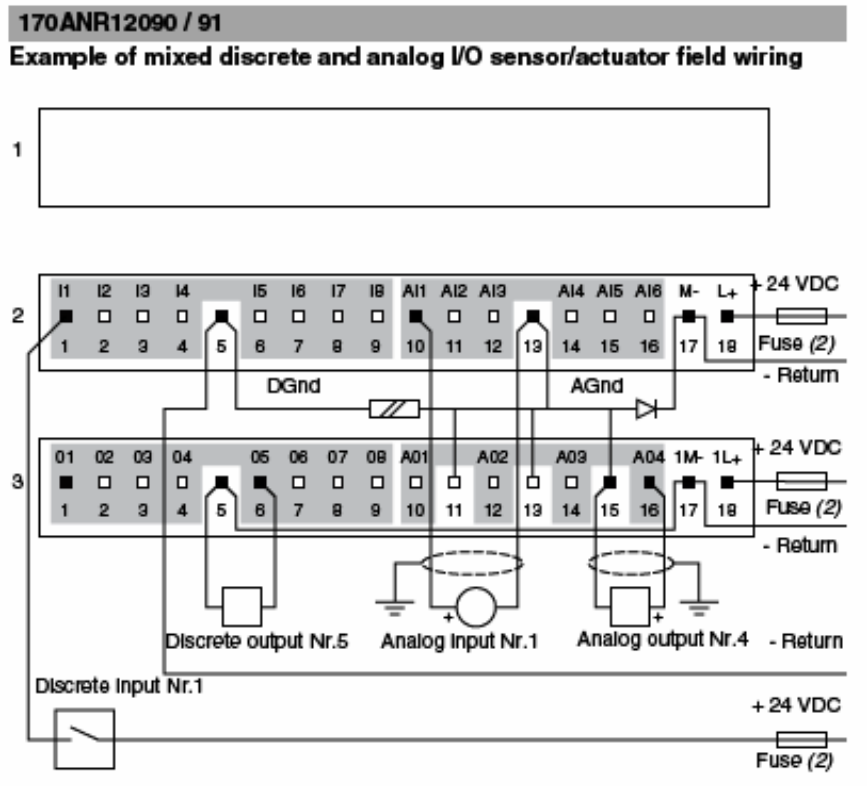
Třídu *ModbusClient* ve jmenném prostoru *Modbuslibrary* v C# jsme rozšířili o funkci `RefreshMomentum`, jejíž parametry jsou pole hodnot analogových a digitálních vstupů a výstupů v daných datových typech. Funkce převede hodnoty výstupů na tvar pro funkci `WriteHoldingRegisters` a zavolá ji. Dále zavolá funkci `ReadHoldingRegisters`. Její výstup zpracuje a naplní pole hodnot vstupů a vrátí je v parametrech. Jedná se tedy o funkci, která usnadňuje práci se zařízením Momentum v podobě zavolání pouze jedné funkce. V této funkci je již nadefinován počet zapisovaných a čtených registrů. Její parametry jsou pouze jednotlivé analogové a digitální vstupy a výstupy.

Aby a data obdržená posláním dotazu o čtení registrů skutečně odpovídala v daném okamžiku stavu vstupně výstupní základny, je nutné nejdříve poslat příkaz se zapsáním registrů. Po tomto příkazu zařízení obnoví registry dle aktuálního stavu základny. V tomto okamžiku můžeme již poslat žádost o vrácení registrů, a tedy zjistíme stavy jednotlivých vstupů a výstupů.

5.3 Vstupně/výstupní základna

Základna 170 ANR 120 90 je vybavena 8 digitálními a 4 analogovými výstupy, 8 digitálními a 6 analogovými vstupy. Rozmístění na svorkovnici a způsob připojení procesní instrumentace jsou zachyceny na obrázku. K základně je připojen ethernetový komunikační modul a vlastní procesor. Tyto jednotky jsou napájeny ze základny. Základna je vybavena indikačními LED diodami pro diskrétní vstupy a výstupy. Stav zařízení indikují LED diody *RUN*, která indikuje bezporuchový chod zařízení, a *ready*, která indikuje, zda zařízení právě komunikuje, tedy nachází se ve stavu ready. Komunikaci přes ethernetový port signalizuje LED dioda *LAN ACT*.

Základnu tvoří tři sloty. První je nevyužit. Druhý je určen pro analogové a diskrétní vstupy. Třetí pro analogové a diskrétní výstupy. Napájení základny se přivede na svorky 17 a 18 ve druhé řadě. Napájení je dále rozvedeno uvnitř přístroje k procesoru a ke komunikačnímu modulu. Napájení výstupů se přivádí na svorky 17 a 18 v třetí řadě. Praktickou realizaci zapojení dle výrobce znázorňuje následující obrázek :



Obr. 5-2 Schéma a zapojení vstupů a výstupů [5]

5.3.1 Technické specifikace

5.3.1.1 Obecné specifikace

Napájecí napětí	24 VDC
Rozsah napájení	20 .. 30 VDC
Proudová spotřeba	max. 400mA
Ochrana diskrétních výstupů	proti přetížení a zkratování
Teplota prostředí pro provoz	0° .. 60°C

Tab. 4-9 Obecné technické specifikace

5.3.1.2 Analogové vstupy

Počet vstupních kanálů	6
Rozsah vstupního kanálu	0 .. 10 VDC
Vstupní impedance	> 1 MΩ
Rozlišení A/D převodu	14 bitů

Chyba převodu	± 1 LSB
Doba převodu	0,75 ms pro všechny kanály
Formát dat	zarovnaný vlevo

Tab. 4-10 Technické specifikace analogových vstupů

5.3.1.3 Analogové výstupy

Počet výstupních kanálů	4
Rozsah výstupního kanálu	0 .. 10 VDC
Odchylka při 25°C	0,4 %
Rozlišení A/D převodu	14 bitů
Chyba převodu	± 2 LSB
Doba převodu	1,2 ms pro všechny kanály
Formát dat	zarovnaný vlevo

Tab. 4-11 Technické specifikace analogových výstupů

5.3.1.4 Diskrétní vstupy

Počet vstupů	8
Stav ON	> 11 VDC
Stav OFF	< 5 VDC
Stav ON	> 6 mA
Stav OFF	< 2 mA
Maximální trvalé napětí	32 VDC
Doba překlopení	max. 1,2 ms
Ochrana proti přetížení	limitováno resistorem, varistorem

Tab. 4-12 Technické specifikace diskrétních vstupů

5.3.1.5 Diskrétní výstupy

Počet výstupů	8
Maximální krátkodobé zatížení	50 VDC za 1 ms
Maximální proud na výstup	0,25 A
Maximální proud na modul	2 A
Maximální klidový proud	0,4 mA při 30 VDC
Maximální krátkodobý proud na výstup	2,5 A za 1 ms

Doba překlopení	max. 1,2 ms
Ochrana proti přetížení	dioda, pojistka Wickmann 2,5A

Tab. 4-13 Technické specifikace diskrétních výstupů

5.4 Shrnutí

Zařízení zpracovává Modbus požadavky klientů a vykonává je na svém fyzickém rozhraní. Ovládá se pomocí příkazů o zapsání a vrácení uchovávacích registrů. O konzistenci vrácených dat ze zařízení a aktuálním stavem vstupů a výstupů základny se stará vytvořená funkce `RefreshMomentum`, která také usnadní práci při programování nadstavbových aplikací.

Jedná se tedy o procesní stanici řídicího systému, se kterým je spojeno pomocí ethernetové sítě. Pro naše potřeby bude jednotka řízena přímo z PC, a to z uživatelské grafické aplikace a z webového rozhraní HTTP serveru. Oba způsoby vyžadují vytvořené knihovny a vytvořené rozhraní pro práci s knihovnou v prostředí C#.

6 VIZUALIZAČNÍ APLIKACE

Řídící vizualizační aplikace umožní operátorovi řídit model jeřábu a přehledně zobrazuje jeho stav. Cyklicky komunikuje se zařízením Momentum TSX a zajišťuje pravdivost vizualizovaného stavu modelu. Také se stará o odezvu modelu na požadavek jeho změny stavu operátorem.

6.1 Popis aplikace

Vizualizační aplikace je uživatelská aplikace běžící na operačních systémech Microsoft Windows 32bit. Je vytvořena v programovacím jazyku C# s využitím dotNET Framework běžící na rozhraní CLR (Common Runtime Language). dotNet Framework je distribuovatelný balíček firmy Microsoft, který poskytuje knihovny pro aplikace vyvinuté v tomto prostředí a který podporuje platformovou nezávislost programového kódu. Vybavenost osobního počítače tímto balíčkem je v dnešní době téměř samozřejmostí. Vlastní aplikace je tvořena jediným souborem. K její správné činnosti je třeba souboru knihovny, adresář pro webové rozhraní a adresář s datovými zdroji aplikace. Datové zdroje tvoří obrázky použité při vizualizaci modelu a ovládacích prvků aplikace.

6.2 Funkce aplikace

Komunikuje se modelem jeřábu pomocí protokolu Modbus TCP, jehož implementace do aplikace je vyřešena pomocí naprogramované knihovny ModbusLibrary.dll. Aplikace tedy používá výstupní funkce z knihovny pro komunikaci se zařízením Modicon TSX Momentum, a tedy pro ovládání modelu jeřábu. Úkolem aplikace je komunikovat se zařízením Momentum TSX, zpracovat jeho odpověď do grafické podoby aktuálního stavu modelu a reagovat na požadavky operátora, co se týče ovládání modelu. Tyto požadavky zpět převést do formy pro použití ve funkci RefreshMomentum.

6.3 Programové řešení

Proces aplikace obsahuje dvě vlákna. První vlákno se stará o běh aplikace a vykreslování okna, prvků formuláře a modelu jeřábu. Druhé vlákno periodicky po daném čase komunikuje se zařízením Momentum TSX, tedy volá funkci knihovny ModbusLibrary `RefreshMomentum()`. Poté zavolá funkci modulu pro vykreslování modelu jeřábu `CountIO()` pro zaktualizování parametrů modelu, a tím pro vykreslení jeho aktuálního stavu.

6.4 Popis modelu jeřábu

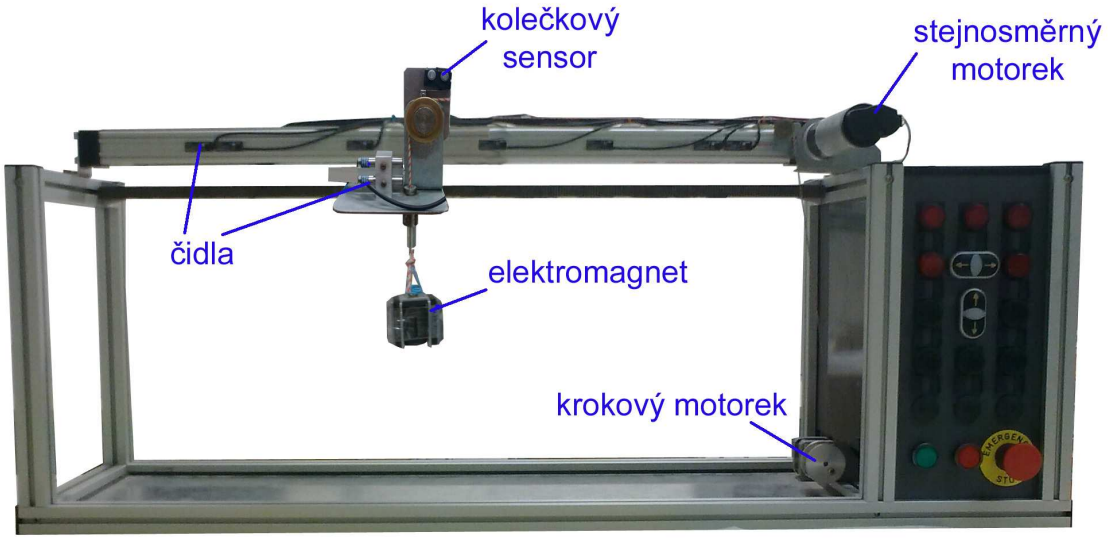
Model jeřábu představuje plně funkční zařízení, které je schopno přesunovat předměty na pracovní ploše. Předměty se nacházejí v jedné ose, přičemž při manipulaci se předmět pohybuje ve vertikální ose. Pro uchopení předmětu při manipulaci se využívá elektromagnetu zavěšeném na laně.

Pohyb elektromagnetu do stran zajišťuje stejnosměrný motorek s lineární převodovkou. Vertikální pohyb elektromagnetu je zajišťuje pomocí lana krokový motorek, který slouží jako naviják. Elektromagnet má dva stavy zapnuto – vypnuto a jeho horní polohu indikuje koncový bezdotykový indukční senzor. Pět indukčních senzorů umístěných na lineární převodovce indikují definované pozice elektromagnetu, přičemž krajní indikují koncové polohy pracovního působiště. Model je zachycen a popsán na obr č.6-1.

6.4.1 Popis komponent modelu jeřábu

Elektromagnet	výstup	+24 VDC .. zapnuto; 0 VDC .. vypnuto
Čidla	vstupy	+24 VDC .. aktivní; 0 VDC .. neaktivní; typ KL3043
Senzor polohy	vstup	připojen přes dělič napětí
Stejnosměrný motorek	výstupy	ovládání rychlosti DC-DC měničem
Krokový motorek	výstupy	připojen k řadiči SD30x, ovládání rychlosti převodníkem
Kolečkový senzor	vstup	senzor pohybu lana, jehož výstupem jsou impulsy

Tab. 6-1 Popis komponent jeřábu



Obr. 6-1 Komponenty jeřábu

6.4.2 Mapování vstupů a výstupů

Následující tabulka zachycuje fyzické zapojení vstupů a výstupů k ovládacím prvkům modelu jeřábu na zařízení Momentum TSX. Jednotlivá označení vstupů a vstupů reprezentují senzory a aktory modelu jeřábu. Každému prvku odpovídá programová proměnná ve struktuře `FIO`. Dle těchto symbolických označení k nim přistupujeme z ovládací aplikace. Mapování vstupů a výstupů vychází z vnitřního uspořádání vstupně výstupní základny (kap. 5.3). Následující tabulky shrnují význam jednotlivých vstupů a výstupů. První sloupec označuje označení daného vstupu či výstupu základny. Druhý sloupec značí svorku na jeřábu, se kterou je vstup či výstup fyzicky propojen. Třetí sloupec značí jméno proměnné ve struktuře `FIO`, která zastupuje daný vstup či výstup. Svorky jsou číslovány zleva. Vnější svorky (zadní) jsou označeny písmenem A, vnitřní (přední) písmenem B.

DO1	20.A	Magnet	Ovládání elektromagnetu
DO2	23.B	PosuvSmer	Přepne směr otáčení motoru posuvu vlevo
DO3	21.A	LanoSmer	Přepne směr otáčení motoru navijáku vpravo
DO4	20.B	LanoActive	Zaktivní driver pro ovládání navijáku

Tab. 6-2 Popis významu digitálních výstupů

DI1	12.B	MagnetCidlo	Čidlo koncové horní polohy elektromagnetu
DI2	10.A	Cidlo1	1. čidlo polohy zleva (levé krajní)
DI3	10.B	Cidlo2	2. čidlo polohy
DI4	11.A	Cidlo3	3. čidlo polohy
DI5	11.B	Cidlo4	4. čidlo polohy
DI6	12.A	Cidlo5	5. čidlo polohy (pravé krajní)
DI7	13.A	KoleckoSensor	Čidlo otáčení ozubeného kolečka

Tab. 6-3 Popis významu digitálních vstupů

AO1	24.B	PosuvRychlost	Ovládání rychlosti posuvu jezdce 0 – 10 V
AO2	21.B	LanoRychlost	Ovládání rychlosti navijáku 0 – 10 V

Tab. 6-4 Popis významu analogových výstupů

AI1	15.A	Pozice	Senzor polohy 0 – 10 V přepočítáno na mm

Tab. 6-5 Popis významu analogových vstupů

6.4.3 Ovládání navijáku

Krokový motorek pro ovládání navijáku je ovládán pomocí řadiče typu MICROCON SD30x. Řadič má tři vstupy. První je pro aktivaci funkce řadiče a může nabývat hodnot +24 VDC nebo 0 VDC. Druhý vstup slouží pro určení směru otáčení motorku. Při hodnotě +24 VDC se motorek otáčí doprava, při hodnotě 0 VDC doleva. Třetí vstup slouží k samotnému ovládání otáček a to pomocí impulsů. Při příchodu impulsu +24VDC řadič pootočí motorkem o jeden krok. Tento vstup je napojen na výstup měniče napětí 0 – 10 V z výstupu Momentum TSX na frekvenci. Tento měnič je tvořen mikrokontrolérem ATtiny s obslužným firmwarem a PWM výstupem. Galvanické oddělení je provedeno optočlenem.

6.4.4 Ovládání posuvu jezdce

Posuv jezdce je zajištěn stejnosměrným motorkem s lineární převodovkou. Rychlost motorku je ovládána napětím. Jmenovité napětí motorku je 12V. Ovládacím výstup z Momentum TSX je analogový výstup s rozsahem 0 – 10V. Ten je vstupem DC – DC měniči, který převede hodnotu na rozsah 0 – 12V a který zároveň slouží jako proudový zesilovač. Na výstup měniče je přímo napojen motorek.

6.4.5 Senzor polohy

Polohový senzor udává absolutní polohu elektromagnetu a je realizován jako potenciometr, který je připojen na převodník napěťové úrovně. Výstup převodníku je 0 – 10 VDC v závislosti na poloze elektromagnetu. Převodník je tvořen děličem napětí, který převede hodnotu z rozsahu 0 – 24V na 0 – 10 V.

6.4.6 Matematický popis

Pro ovládání jednotlivých komponent jeřábu je třeba přepočítat hodnotu analogového vstupu či výstupu o velikosti bezznaménkového slova, tedy 0 .. 65 535 na fyzikální veličiny odpovídající významu hodnoty. Tento přepočet je třeba provést pro senzor polohy, pro ovládání rychlosti motorku posuvu jezdce a pro ovládání rychlosti navijáku. S využitím matematického popisu můžeme zajistit synchronní rychlost otáčení navijáku a posuvu jezdce a tím docílit, aby elektromagnet se pohyboval při pojezdu do stran ve stejné výšce nad podložkou.

Přepočet hodnoty senzoru polohy

Rozsah hodnot z analogového vstupu, na který je připojen senzor polohy je 0 .. 32766, jak vyplývá ze specifikace zařízení Momentum TSX (kap. 5.1.1.3). Tabulka zjištěných údajů :

Poloha od relativního počátku *l[mm]*	**Odpovídající analogová hodnota *h[-]***
0	6950
520	28650

Tab. 6-6 Naměřené hodnoty senzoru polohy

Ze zjištěných údajů lze pomocí lineární aproximace určit převodní vztah :

$$l[mm] = \frac{520}{28650 - 6950}(h - 6950) \tag{1}$$

Přepočet hodnoty rychlosti posuvu

Rozsah hodnot z analogového výstupu, na který je připojen DC – DC měnič pro ovládání rychlosti motorku posuvu jezdce, je 0 .. 32766, jak vyplývá ze specifikace zařízení Momentum TSX (kap. 5.1.2.6). Tabulka zjištěných údajů :

Naměřená rychlost jezdce *v[mm/s]*	**Odpovídající analogová hodnota *h[-]***
13,0	6000
30,6	10000
52,0	15000

Tab. 6-6 Naměřené hodnoty rychlosti jezdce

Ze zjištěných údajů lze pomocí lineární aproximace určit převodní vztah :

$$v[mm/s] = \frac{52{,}0 - 13{,}0}{15000 - 6000}(h - 10000) + 30{,}6 \tag{2}$$

Přepočet hodnoty rychlosti navijáku

Rozsah hodnot z analogového výstupu, na který je připojen napětím řízený oscilátor pro ovládání rychlosti krokového motorku navijáku, je 0 .. 32766, jak vyplývá ze specifikace zařízení Momentum TSX (kap. 5.1.2.6). Zjištěné údaje :

Naměřená rychlost navijáku *v[mm/s]*	**Odpovídající analogová hodnota *h[-]***
54,8	1680
94,4	2000
135,1	2500

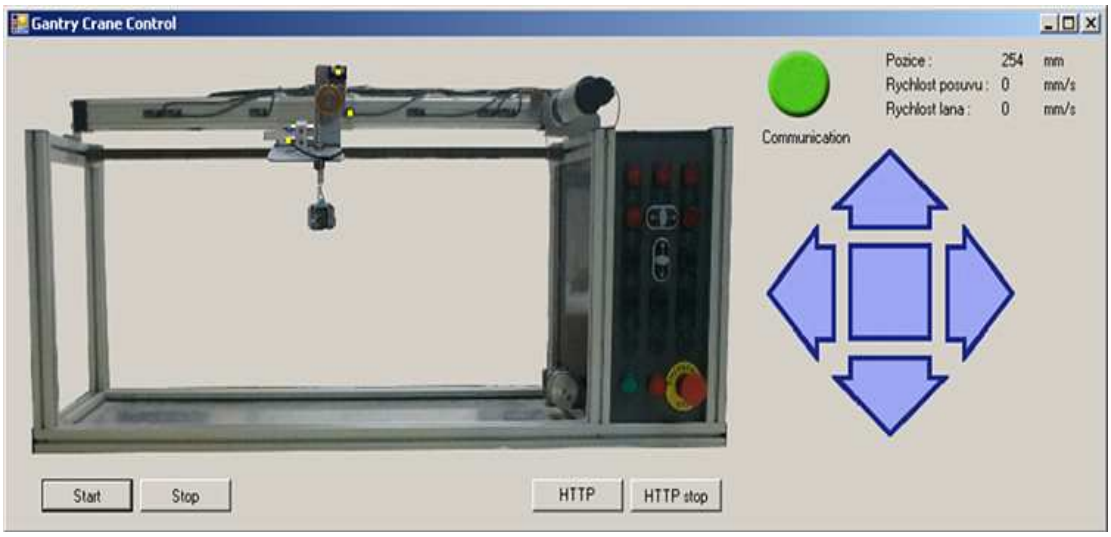
Tab. 6-6 Naměřené hodnoty rychlosti navijáku

Ze zjištěných údajů lze pomocí lineární aproximace určit převodní vztah :

$$v[mm/s] = \frac{135{,}1 - 13{,}0}{2500 - 1680}(h - 2000) + 94{,}4 \qquad (3)$$

6.5 Ovládání

Ovládání se provádí pomocí šipek, které představují směr pohybu elektromagnetu. Prostřední tlačítko tento elektromagnet ovládá, tedy přepíná stav elektromagnetu zapnuto – vypnuto. Při držení šipky požadovaného směru se pohybuje magnet v tomto směru konstantní definovanou rychlostí. Při požadavku přejetí mezí, které jsou dány geometrickým uspořádáním a technickým řešením modelu, je tento požadavek automaticky anulován. Tato funkce vyžaduje, aby byla řídící aplikace zkalibrovaná s daným připojeným modelem jeřábu. Uživatel je informován o správné komunikaci se zařízením Momentum TSX pomocí signalizující kontrolky. Také je informován o stavu modelu, tedy o pozici jezdce, aktuální rychlosti posuvu jezdce a navijáku. Na následujícím obrázku je screenshot aplikace.



Obr. 6-1 Screenshot aplikace Gantry Crane Control

6.6 Grafika

Formulář aplikace je vytvořen v prostředí Microsoft Visual Studio pomocí standardních prvků, jako jsou tlačítka, editační políčka a grafická pole. Vizualizace modelu jeřábu využívá sady funkcí knihovny Drawing s podporou GDI+ (Graphical Device Interface). Pomocí těchto funkcí lze implementovat jednoduchou 2D grafiku do aplikace. Lze vykreslovat geometrické útvary, text a bitmapy. Knihovna disponuje funkcemi pro libovolné nastavení parametrů vykreslování, jako jsou tloušťka a barva čar, pattern a

barva výplně útvarů, aj. Pro vykreslení stavu modelu jeřábu je toto zobrazení informativně dostatečné. V případě potřeby zobrazení kvalitnějšího designu v 3D prostoru by bylo nutné použít jiné knihovny, např. OpenGL nebo DirectX. Využití knihoven pro tuto aplikaci je ale zbytečné plýtvání systémovými prostředky a navíc se plně nevyužije potenciál, kterým knihovny disponují.

GDI je součást operačního systému Microsoft Windows a slouží k reprezentaci grafických objektů a jejich transformací do výstupních zařízení jako jsou monitory či tiskárny. GDI neslouží k vykreslování formuláře a jeho prvků. O to se stará knihovna user32.dll tedy GUI (Graphical User Interface), jenž je nadstavbou knihovny GDI. GDI udržuje v systému aktuální DC (Device Context), který je používán k definování atributů textu a obrázků, které jsou zobrazovány na obrazovce.

S příchodem Microsoft Windows XP se založil nový subsystém GDI+, který má oproti GDI vylepšené 2D grafické prostředí a má implementovány funkce 2D anti-aliasing, alpha blending, aj. Také navíc podporuje více grafických formátů jako jsou např. JPEG a PNG. Počínaje operačním systémem Microsoft Windows Vista běží GDI a GDI+ na novém enginu s DWM jádrem, a proto mají všechny grafické změny rychlejší odezvu.

Kreslení pomocí GDI je možno využít pomocí volání funkcí z knihovny Drawing. Tyto funkce se umístí do události vykreslovací komponenty `OnPaint()`. Tím je zajištěno, že se funkce zavolají při každém požadavku o překreslení (změna velikosti, maximalizace, zviditelnění).

Při vykreslování pohybujících se objektů je nutno odstraňovat relikty objektů na předchozích pozicích. To zajistíme zavoláním metody `Invalidate()` nad kreslící komponentou, která vyvolá událost `OnPaint()`. Princip vykreslování pohybujících se objektů spočívá v rychlém vykreslováním objektů s měnícím se umístěním, aby se pozorovateli zdálo, že objekt vykonává plynulý pohyb.

Komponenta `PictureBox`, kterou využíváme jako kreslící plochu, ja primárně určena pro zobrazování grafických zdrojů. Toho využíváme pro zobrazení samotného modelu jeřábu, jakožto základní obraz. Pomocí funkcí z knihovny Drawing vykreslujeme jednotlivé komponenty, jejichž poloha je závislá na obdržených datech ze zařízení Momentum TSX. Využité funkce :

`FillEllipse(Brush, Rectangle)` – vykreslí elipsu v daném čtverci

`DrawImage(Image, Rectangle)` – vykreslí daný obrázek v daném čtverci

Vykreslení aktuálního modelu jeřábu tedy zajišťuje sada funkcí umístěná v události `OnPaint` grafické komponenty, která slouží jako plátno. Tato událost je volána funkcí `Invalidate()`.

6.7 Pozice elektromagnetu

Pro zobrazení stavu modelu jeřábu je nutno získat prostřednictvím zařízení Momentum TSX hodnoty z jednotlivých čidel polohy. Získávání informací tedy závisí na vybavenosti modelu snímači polohy jednotlivých komponent modelu. Pro určení polohy elektromagnetu ve vertikálním směru ale není žádný přímý snímač polohy namontován. Pro určení pozice elektromagnetu můžeme využít následující nepřímé metody :

- vertikální polohu spočítáme s použitím nastavené rychlosti navijáku
- vertikální polohu určíme pomocí signálu z ozubeného senzoru a koncového čidla

Vertikální pozici elektromagnetu určíme zkombinováním obou metod. Tedy pozice je vypočítávána z údaje nastavené rychlosti. Horní poloha elektromagnetu, kdy je elektromagnet relativně maximálně vytažen nahoru, je snímána koncovým čidlem. Dolní poloha elektromagnetu, kdy je elektromagnet plně spuštěn dolů, je snímána přerušením otáček ozubeného senzoru. Toto přerušení otáček symbolizuje, že elektromagnet narazil při spouštění na překážku. Pro zjištění, že se elektromagnet nachází v mezní dolní poloze, je zapotřebí znát geometrické vlastnosti modelu, tedy maximální výšku vysunutí elektromagnetu, a je třeba přihlédnout k vypočtené poloze elektromagnetu dle první metody.

Určení pozice elektromagnetu při absenci přímého měřícího aparátu je zatíženo chybou definovanou použitím nepřímé měřící metody a její chybou měření, která je také dána konstrukčními vlastnostmi snímacího ústrojí. Odchylka rekonstruované vertikální pozice elektromagnetu od její skutečné pozice může být výrazná také díky vnějšímu zásahu do modelu jeřábu, či jiné nestandardní situaci, vzniklé působením poruch.

Jelikož je ale tento údaj brán jako informativní a slouží jako ukázka možností vizualizační aplikace, je tento způsob jeho získání dostačující. Při šetrném ovládání modelu jeřábu a při bezporuchovém stavu celého systému bude jeho hodnota s určitou chybou platná.

6.8 Číselné údaje

Všechny informace o stavu modelu v číselné podobě jsou pouze informativní. Jejich měření bylo provedeno v amatérských podmínkách, a tedy jsou zatíženy poměrně velkou chybou měření. Jejich měření nebylo opakováno, a proto je chyba měření zvětšena o náhodné vlivy, které diskreditují měření. Důvod jejich použití je informování orientačních hodnot sledovaných parametrů a ukázka možností vizualizační aplikace jako operátorského centra.

6.9 Závěr

Aplikace je ukázkou jednoduchého řídícího centra pro model jeřábu. Disponuje základními funkcemi, jako jsou zobrazování stavu modelu jeřábu pomocí jednoduché 2D grafiky a základní ovládání pohybu a elektromagnetu. Hlavní funkce aplikace jsou závislé na funkceschopnosti knihovny pro komunikaci se zařízením Momentum TSX, s jeho bezporuchovým chodem a akceschopností.

Vylepšením vizualizační aplikace by bylo zpříjemnění uživatelského prostředí, což je ale zcela závislé na požadavcích operátora. Dále by bylo možné přidat další sledované údaje, aby operátor měl ucelené informace o stavu procesu. Bylo by možné naimplementovat předdefinované algoritmy pro práci s modelem, tzn. naprogramovat algoritmy, které usnadní práci se zařízením, např. najíždění na určitou pozici, automatické uchopování předmětu nebo již plně automatizovanou úlohu, která řeší daný problém.

7 HTTP SERVER

Posledním požadavkem zadání bylo implementovat do aplikace webový server poskytující základní údaje o stavu modelu. Z důvodu, že server bude ovládán z aplikace a také s touto aplikací bude sdílet data, jsme se rozhodli vytvořit vlastní HTTP server jen za použití komponent pro komunikaci pomocí TCP protokolu.

7.1 Popis a implementace

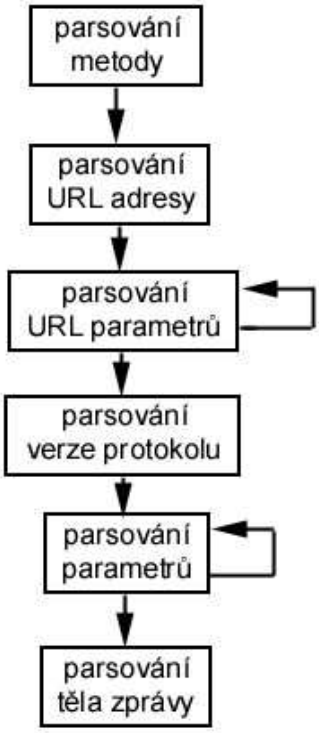
Webový server je naprogramován jako samostatná třída, která po zinicializování pracuje v samostatném vlákně, takže neruší chod vlastního programu. Je tedy nutné vytvořit objekt této třídy. Dále pomocí funkce `Start()` se spustí samostatná webová služba aplikace a pomocí funkce `Stop()` se její činnost ukončí. Webový server je spuštěn na daném počítači, tedy na adrese localhostu.

7.2 Princip činnosti

Server pracuje jako stavový automat. To znamená, že po přijetí žádosti od připojeného klienta tuto zprávu zpracuje pomocí algoritmu stavového automatu, připraví odpověď, odešle ji klientovi a ukončí spojení. Odpověď obsahuje v případě nesprávné žádosti, či jiné chyby vzniklé parsováním žádosti, číslo a popis vzniklé chyby HTTP protokolu.

Po zavolání metody `Start()` se vytvoří vlákno z metody `ListenForClients()`, které slouží k naslouchání serveru na portu. Po připojení klienta, server spojení přijme a vytvoří vlákno z metody `HandleClient()`. Vytvořené vlákno přijme požadavek klienta a pomocí stavového automatu ji parsuje a naplní strukturu `HTTPRequest`. Poté vlákno zavolá metodu `PrepareResponse()`, které vyhodnotí data ve struktuře HTTPRequest a dle požadavku naplní strukturu `HTTPResponse`. V případě žádosti o stránku informačního panelu modelu jeřábu zavolá metoda metodu klienta pro komunikaci s mateřskou aplikací. V případě výskytu chyby odešle zprávu o chybě dle HTTP protokolu. Po odeslání odpovědi server komunikaci ukončí a tím je vlákno pro obsluhu klienta ukončeno.

Činnost stavového automatu pro zpracování žádosti, tedy naplnění struktury `HTTPRequest`, ilustruje obrázek č.7-1.



Obr. 7-1 Algoritmus parsování požadavku klienta

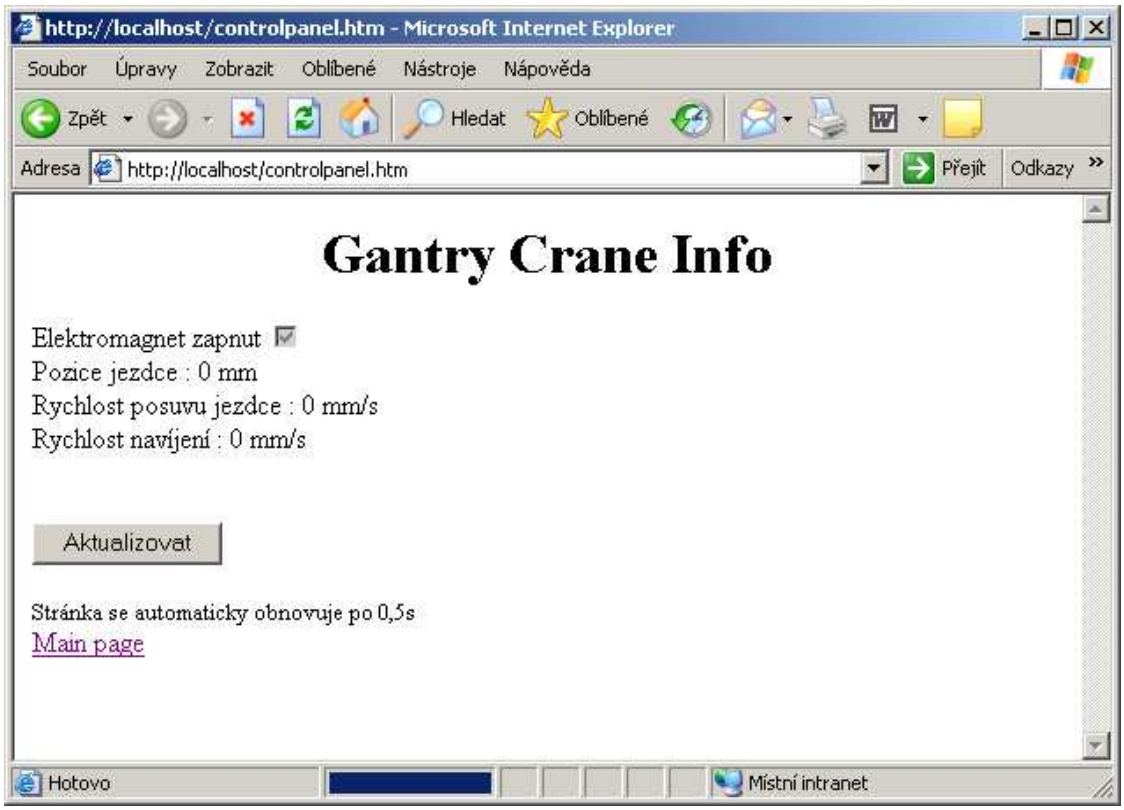
7.3 Použité komponenty

HTTP server využívá objekt z knihovny .NET Frameworku `TcpListener`, jenž slouží jako abstraktní vrstva v ovládání socketů pro komunikaci s klienty. Tento objekt je nutno nejdříve zinicializovat. Po zavolání metody `Start()` se uvede do stavu naslouchání na určeném portu. Po vykonání blokující metody `AcceptTcpClient()` naváže spojení s klientem a vytvoří objekt `TcpClient`. Samotná výměna zpráv probíhá prostřednictvím objektu `NetworkStream`. Metoda `Read()` zprávu přečte, metoda write zprávu pošle. Po zaslání odpovědi server ukončí činnost objektu `TcpClient` zavoláním metody `Close()`.

7.4 Webové rozhraní modelu

Webové rozhraní modelu jeřábu je tvořeno stránkami v podobě souborů ve složce Web, která musí být přítomna v adresáři se spouštěcím souborem. Webové prostředí se skládá ze dvou stránek. První dává informaci, že se uživatel nachází ve webovém rozhraní, a je to startovací stránka, která se jmenuje *index.htm*. Pomocí odkazu Informační panel se uživatel dostane ke druhé stránce *controlpanel.htm*, na které jsou informace o stavu modelu jak je pozice jezdce, rychlost posuvu jezdce, rychlost navíjení a informace, zda

je elektromagnet zapnut. Tato stránka se automaticky obnovuje po 500ms a její design je zachycen na následujícím obrázku.



Obr. 7-2 Screenshot webového rozhraní

7.5 Shrnutí

Vytvořený webový server poskytuje webový prostor pro webové rozhraní modelu jeřábu. Toto rozhraní dává základní informace o stavu modelu jeřábu. Server je ovládán z mateřské aplikace a běží samostatně, aniž by rušil ostatní činnost aplikace. Při potřebě řídit jeřáb z webového rozhraní by bylo nutné vytvořit další stránku, která by se obnovovala při potřebě ovládat. Další variantou by bylo implementovat do serveru jednoduchou formu CGI. Pomocí této technologie by bylo možné efektivně řídit a zjišťovat aktuální stav modelu.

8 ZÁVĚR

Tato práce se zabývala řízením daného modelu jeřábu pomocí osobního počítače. Procesní instrumentace modelu je napojena přes svorkovnici na vstupy a výstupy zařízení Modicon Momentum TSX. Toto zařízení je spojeno s osobním počítačem ethernetovým kabelem a komunikuje otevřeným protokolem.

Pro řízení této soustavy z osobního počítače jsme vytvořili DLL knihovnu, která tvoří aplikační vrstvu komunikačního modelu ISO/OSI. Funkce z této knihovny jsme implementovali do programovacího prostředí C# a v tomto prostředí vytvořili dvě uživatelské aplikace. První slouží pro ukázku možností knihovny ve směru funkčnosti a obsahu. To znamená, že uživatel si může spustit virtuální server, k němu se připojit pomocí klienta a posílat požadavky dle Modbus protokolu. Na kartě serveru pak vidí aktuální stav dat na serveru. Druhá aplikace byla již předmětem zadání a slouží jako vizualizační aplikace pro ovládání modelu jeřábu. Uživatel vidí poměrně realistický obrázek vypovídající o stavu modelu. Pomocí ovládacích šipek může ovládat pohyb jezdce ve všech čtyřech možných směrech. Aplikace je intuitivní a zahrnuje i ovládání webového rozhraní ve smyslu zapnutí a vypnutí. Webové rozhraní je naimplementováno pomocí vestavěného modulárního webového serveru, který informuje o stavu modelu. O stavu modelu může být informováno více uživatelů a to právě díky webovému serveru, na který se jednoduše připojí pomocí webového prohlížeče.

Tato práce nám ukázala možnosti ovládání procesní instrumentace prostřednictvím daného zařízení přímo z osobního počítače, a to i na větší vzdálenosti od daného procesu. Dává nám programové vybavení, pomocí kterého bude vytváření uživatelských a zobrazovacích aplikací snazší a komfortnější.

Literatura

[1] MODBUS – IDA Modbus Application Protocol Specification v.1.1b [on line], December 2006, 51 p., Dostupné na URL: http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf

[2] Ronešová, A. Přehled protokolu Modbus [on line], květen 2005, 20 str., Dostupné na URL: http://home.zcu.cz/~ronesova/bastl/files/modbus.pdf

[3] Wikipedie Modbus [on line], listopad 2010, Dostupné na URL: http://cs.wikipedia.org/wiki/Modbus

[4] MSDN online help [on line], listopad 2010, Dostupné na URL: http://msdn.microsoft.com/library

[5] Schneider - Electric Ethernet Communication Adapter 170 ENT 110 00 [on line], User Guide 870 USE 112 00, v. 1.0, 30 p., July 1998, Dostupné na URL: http://v1.graybar.com/automation/ga_manuals/Hardware/TsxMomentum/870USE11200%20%2807-98%29%20170ENT11000%20Ethernet%20Communication%20Adapter.pdf

[6] Schneider – Electric Modicon TSX Momentum I/O Base [on line], User Guide 870 USE 002 00, v. 4.0, p. 511 – 534, February 2003, Dostupné na URL: http://v1.graybar.com/automation/ga_manuals/Hardware/TsxMomentum/870USE00200%20Ver.4%20Chpt%2031%20&%2032%20-%20170%20ARM%20370%2030.pdf

[7] Marshall J., HTTP Made Really Easy [on line], August 1997, Dostupné na URL: http://www.jmarshall.com/easy/http/

[8] Grothoff Ch., GNU libmicrohttpd [on line], 2007, Dostupné na URL: http://www.gnu.org/software/libmicrohttpd/

[9] Wrox, Professional C# - Graphics with GDI+ [on line], September 2001, Dostupné na URL: http://www.codeproject.com/KB/books/1861004990.aspx

Seznam příloh

Příloha 1. Mini disk CD se zdrojovými projekty aplikací :

- CraneControl – adresář s projektem ovládání jeřábu v jazyce C#
- ModbusControl – adresář s projektem aplikace Modus v jazyce C#
- ModbusLibrary – adresář s projektem Modbus knihovny DLL v jazyce C